Logical Foundations of Object-Oriented and Frame-Based Languages

Michael Kifer [*] Georg Lausen [†] James Wu [‡]

Abstract

We propose a novel formalism, called *Frame Logic* (abbr., F-logic), that accounts in a clean and declarative fashion for most of the structural aspects of object-oriented and frame-based languages. These features include object identity, complex objects, inheritance, polymorphic types, query methods, encapsulation, and others. In a sense, F-logic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming. F-logic has a model-theoretic semantics and a sound and complete resolution-based proof theory. A small number of fundamental concepts that come from object-oriented programming have direct representation in F-logic; other, secondary aspects of this paradigm are easily modeled as well. The paper also discusses semantic issues pertaining to programming with a deductive object-oriented language based on a subset of F-logic.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Languages—*query languages*; I.2.3 [**Artificial Intelligence**]: Deduction and theorem proving—*deduction, logic programming, nonmonotonic reasoning*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical logic—*logic programming, mechanical theorem proving*

General Terms: Languages, Theory

Additional Key Words and Phrases: Object-oriented programming, frame-based languages, deductive databases, logic programming, semantics, proof theory, typing, nonmonotonic inheritance

Journal of the Association for Computing Machinery, May 1995

[*]Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, U.S.A. Email: kifer@cs.sunysb.edu. Work supported in part by NSF grants DCR-8603676, IRI-8903507, and CCR-9102159.

[†]Institut für Informatik, Universität Freiburg, Rheinstrasse 10-12, 79104 Freiburg, Germany. Email: lausen@informatik.uni-freiburg.de

[‡]Renaissance Software, 175 S. San Antonio Rd., Los Altos, CA 94022, U.S.A. Email: wu@rs.com. Work supported in part by NSF grant IRI-8903507.

Contents

1 Introduction

In the past decade, considerable interest arose in the so-called *object-oriented* approach, both within the database community and among researchers in programming languages. Although "the object-oriented approach" is only a loosely defined term, a number of concepts, such as complex objects, object identity, methods, encapsulation, typing, and inheritance, have been identified as the most salient features of that approach [13, 96, 113, 107, 10].

One of the important driving forces behind the interest in object-oriented languages in databases is the promise they show in overcoming the, so called, *impedance mismatch* [74, 113] between programming languages for writing applications and languages for data retrieval. At the same time, a different, *deductive* approach gained enormous popularity. Since logic can be used as a computational formalism *and* as a data specification language, proponents of the deductive programming paradigm have been arguing that this approach overcomes the aforesaid mismatch problem just as well. However, in their present form, both approaches have shortcomings. One of the main problems with the object-oriented approach is the lack of logical semantics that, traditionally, has been playing an important role in database programming languages. On the other hand, deductive databases rely on a flat data model and do not support data abstraction. It therefore can be expected that combining the two paradigms will pay off in a big way.

A great number of attempts to combine the two approaches has been reported in the literature (*e.g.*, [1, 2, 3, 14, 17, 18, 35, 58, 60, 68, 66, 73, 93, 11]) but, in our opinion, none was entirely successful. These approaches would either seriously restrict object structure and queries; or they would sacrifice declarativity by adding extra-logical features; or they would omit important aspects of object-oriented systems, such as typing and inheritance.

In this paper we propose a formalism, called *Frame Logic* (abbr., F-logic), that achieves *all* of the goals listed above and, in addition, it is suitable for defining, querying, and manipulating database schema. F-logic is a *full-fledged* logic; it has a model-theoretic semantics and a sound and complete proof theory. In a sense, F-logic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming.

Apart from object-oriented databases, another important application of F-logic is in the area of frame-based languages in AI [42, 80], since these languages are also built around the concepts of complex objects, inheritance, and deduction. It is from this connection that the name "Frame Logic" was derived. However, most of our terminology comes from the object-oriented parlance, not from AI. Thus, we will be talking about objects and attributes instead of frames, slots, and the like.

For reasoning about inheritance and for knowledge base exploration, a logic-based language would be greatly aided by higher-order capabilities. However, higher-order logics must be approached with caution in order to preserve desired computational properties. In the past, a number of researchers suggested that many useful higher-order concepts of knowledge representation languages can be encoded in predicate calculus [48, 77]. From the programmer's point of view, however, encoding is not satisfactory, as it gives no *direct* semantics to the important higher-order constructs and it does not retain the spirit of object-oriented programming. In contrast, F-logic represents higher-order and object-oriented concepts directly, both syntactically and semantically.

This work builds upon our previous papers, [58, 55, 60], which in turn borrowed several important ideas from Maier's O-logic [73] (that, in its turn, was inspired by Aït-Kaci's work on ψ-terms [7, 6]).

In [58, 60], we described a logic that adequately covered the structural aspect of complex objects but was short of capturing methods, types, and inheritance. The earlier version of F-logic reported in [55] was a step towards a higher-order syntax. In particular, it supported querying database schema, and structural inheritance was built into the semantics. Nevertheless, this early version of F-logic had several flaws.

One of the problems was that the universe of objects was required to form a lattice, which turned out to be impractical for a logic-based language. Another problem was that semantics was more appropriate for modeling object types than object states. All these problems have been rectified in the present paper and the logic was extended to accommodate typing, non-monotonic inheritance, and other features.

One aspect of knowledge based systems that is not dealt with here is *database dynamics*, which involves changes to the internal state of the objects. Our experience shows that database dynamics is orthogonal to the *structural* aspects of object-oriented systems, which are the focus of this paper. There has been extensive work on formalizing updates within logic. The reader is referred to the recent work on *Transaction Logic* [21, 22, 23] for a comprehensive discussion of the problem, for an overview of the related work in the field, and for solutions to many of the previously outstanding problems. In Section 17.4, we briefly touch upon the issue of extending F-logic in the direction of Transaction Logic, a combination that captures the dynamic aspects of object-oriented languages, including updates and methods with side effects.

An important aspect of F-logic is its *extensibility*—it can be combined with a broad range of other specialized logics. In Section 17, we outline two such combinations: one with HiLog [34] and one with Transaction Logic [21, 22, 23]. Another possible candidate is *Annotated Predicate Logic* [20, 56, 57], which is a logic for reasoning with inconsistent and uncertain information. This extensibility places F-logic in the center of a powerful unifying formalism for reasoning about data and knowledge.

This paper is organized as follows. Section 2 discusses the relationship between the object-oriented paradigm and the relational paradigm. Section 3 is an informal introduction to some of the main features of the logic. In Sections 4, 5, and 6, we describe the syntax and the semantics of F-logic. Section 7 discusses various semantic properties of the logic. Sections 8 through 11 develop a proof theory. Section 12 demonstrates the modeling power of F-logic via a number of non-trivial examples. Section 13 discusses typing. In Section 14 we introduce the notion of encapsulation. Here we propose to view encapsulation as a kind of type-correctness policy and show that semantically rich encapsulation disciplines can be represented in this way. Section 15 presents a semantics for inheritance. An array of issues in data modeling, such as complex values, versions control, and path expressions, is covered in Section 16. Further extensions to the logic are discussed in Section 17. In Section 18 we provide a retrospective view on the internal structure of F-logic and relate this logic to classical predicate calculus. Section 19 concludes the paper.

2 A Perspective on Object-Oriented vs. Declarative Programming

A number of researchers argued that object-oriented languages are fundamentally different from (and even incompatible with) other paradigms, especially with logic programming [102, 75]. This point of view was reflecting disappointment with the lack of early success in formalizing a number of central aspects of object-oriented programming. Compounding the problem was the lack of a framework in which to compare various approaches, so that differences and similarities could not be seen in a common framework. In this section, we make an attempt to clarify these issues.

It has become customary to define the notion of object-orientation via a long list of properties an object-oriented system must possess. Some works (*e.g.*, [10]) further divide these properties into "mandatory" and "optional." While these surveys helped identify the issues, defining a paradigm by enumerating its features seems somewhat perfunctory. Indeed, what is so fundamental about the features that allegedly set the object-oriented approach apart from the relational paradigm? And why is it that some features are claimed to be "more object-oriented" than others?

We believe that the answer lies in the way various languages represent data. In our view, one central feature of the relational model is that data are conceptually grouped around properties. For instance, information regarding a given person would normally be scattered among different relations, such as *employee*, *manager*, and *project*, each describing different properties attributed to persons. On the other hand, pieces of data that describe different persons via the same property, *e.g.*, projects they work on, would be grouped together in a single relation, such as *project*. In contrast, object-oriented representation seeks to group data around objects (again, at the conceptual level). According to this philosophy in language design, the user can access—directly or by inheritance—all public information about an object once a "handle" to that object is obtained. This handle is usually referred to as *physical object identity*—an implementational notion that also has a conceptual counterpart, which we call *logical object identity*. Logical oid's were introduced in [58, 60] and, subsequently, were utilized in [35, 16] and other works.

The concept of object identity was widely discussed in database literature. An early attempt to bring this notion into the fold of a logical theory was reported by Maier [73]. Reflecting on this work, Ullman [102] concluded that the very concept of object identity is incompatible with logic. We feel that both, Maier's difficulties and Ullman's pessimism, arise from a confusion that can be traced to lumping physical and logical object identities together in one notion.

Having decided to group data around objects, it is only natural to try and tap into the potential of such representation by making use of class hierarchies, inheritance, typing, and so on. The difference between "optional" and "mandatory" properties put forward in [10] now becomes a subjective matter of what one perceives to be the most important features of this style of data representation.

To sum up, our view is that the main difference between object-oriented and relational languages is in their *data representation paradigm*.

Another classification dimension for programming languages is their *programming paradigm*, which consists mainly of the following three categories: procedural, functional, and deductive. Figure 1 shows how various languages may fit into this framework. Our contention is that the misconception about the incompatibility of deductive and object-oriented languages comes from overlooking the fact that the two classification axes of Figure 1 are really *orthogonal*. What is commonly referred to as "deductive database languages," is simply a class of languages characterized by the flat, relational data model and the deductive programming paradigm. In contrast, most of the systems that are perceived as "object-oriented" are procedural. Only recently, a number of upward-compatible object-oriented logics have been introduced [35, 58, 55], which made it clearer how the perceived incompatibility gap could be bridged. The present work continues this line of research, closing many gaps and rectifying the flaws of our earlier attempts [58, 55].

We also note that practical considerations often force different paradigms to co-exist under one roof. For instance, Prolog [36] has control primitives that give it procedural flavor. Logical object-oriented languages [58, 35, 55, 3] allow grouping of data around properties (*i.e.*, relationally) as well as around objects. IQL [3] relies on a small number of procedural features. Pascal-R [94] offers declarative access to data, although the overall structure of the language is procedural.

We believe that the future belongs to multi-paradigm languages, so the aforementioned "impurity"

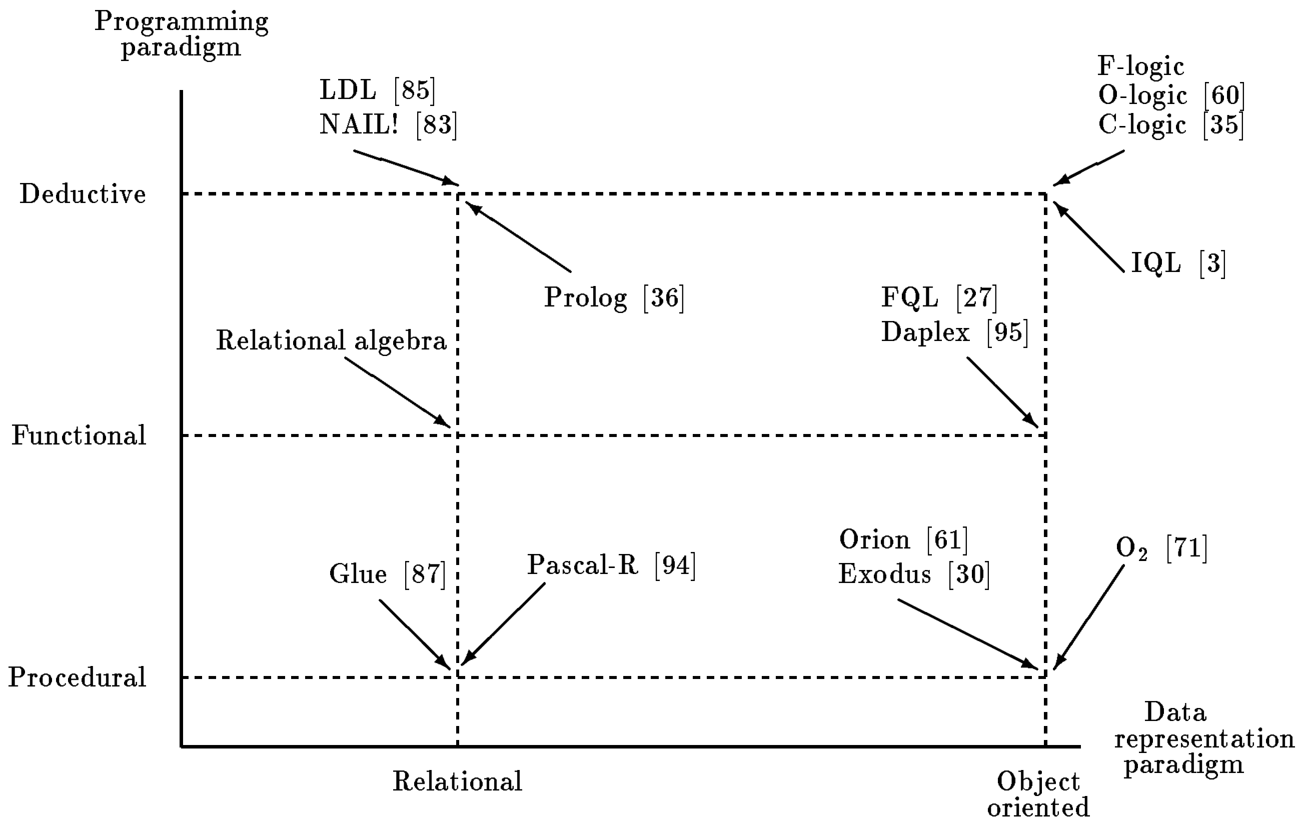


Figure 1: A Classification of Database Programming Languages

is not necessarily a drawback. However, *clean* integration of various paradigms is an important research issue. It will be clear from the present paper that, in the deductive domain, relational and object-oriented data representation paradigms go hand-in-hand. However, we are unaware of any clean solution to the integration problem for procedural object-oriented and relational languages; likewise, there is no generally agreed upon framework for integrating functional and deductive paradigms.

3 F-logic by Example

The development of F-logic is guided by the desire to capture, in a logically clean way, a number of knowledge representation scenarios. Several of the most salient features of F-logic are described in this section by way of an example.

The IS-A Hierarchy

Figure 2 shows part of a hierarchy of classes and individual objects, where solid arcs represent the subclass relationship and dotted arcs represent class membership. This hierarchy asserts that *faculty* and *manager* are subclasses of *empl*; *student* and *empl* are subclasses of *person*; "*Mary*" and "*CS*" are members of the class *string*; and *mary* is a *faculty*, while *sally* is a member of the class *student*. Note that classes are

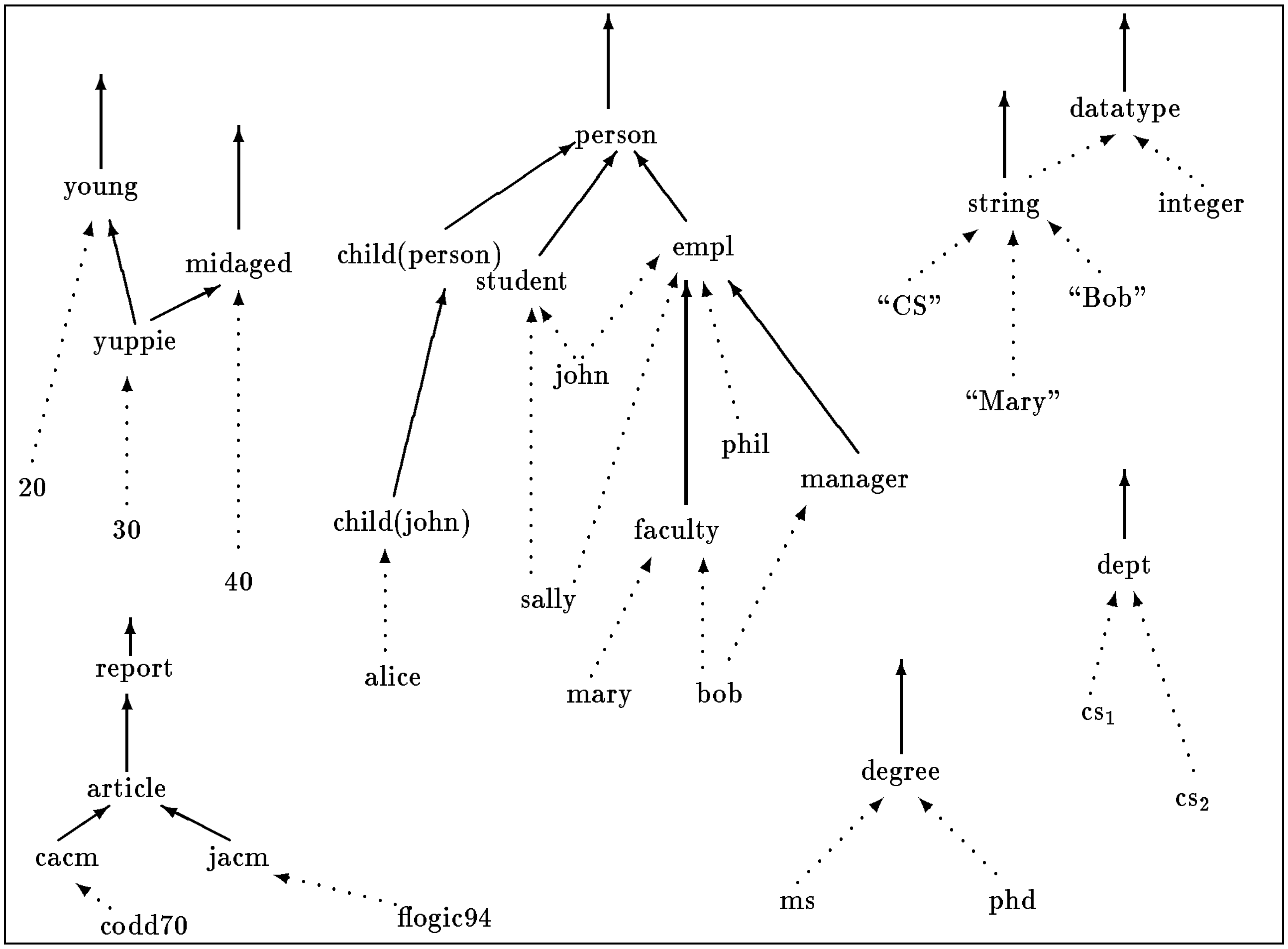


Figure 2: Part of an IS-A Hierarchy

reified, *i.e.*, they belong to the same domain as individual objects. This endows F-logic with a great deal of uniformity, making it possible to manipulate classes and objects in the same language. In particular, when viewed as an object, a class can be a member of another class. For instance, in Figure 2, the classes *string* and *integer* are members of the class *datatype*.

In the actual syntax of F-logic, we use ":" to represent class membership and "::" to denote the subclass-relationship. Thus, for instance, the hierarchy of Figure 2 is recorded as shown in Figure 3. Here a statement such as *empl* :: *person* says that *empl* is a subclass of *person*; *john* : *empl* says that *john* is an instance (*i.e.*, a member) of the class *empl*, and so on.

Notice that objects are denoted via the usual first-order terms, *john*, *empl*, *person*, etc. At this point, we do not assume that every object has a *primary* class (the unique, lowest class in the class hierarchy where the object is a member) or that the class hierarchy is *discrete* (every path that connects a pair of classes goes through only a finite number of intermediate classes). Although primary classes and discrete hierarchies are assumed by many object-oriented systems, they do not seem to be fundamental enough to deserve canonization in a logic. Section 16.5 shows how these features can be represented in F-logic.

$empl :: person$	$yuppie :: young$
$student :: person$	$20 : young$
$faculty :: empl$	$30 : yappie$
$child(person) :: person$	$40 : midaged$
$john : student$	$\text{"}CS\text{"} : string$
$john : empl$	$\text{"}Bob\text{"} : string$
$cs_1 : dept$	$alice : child(john)$
$\ddots \quad \ddots$	$\ddots \quad \ddots$

Figure 3: F-logic Representation of the IS-A Hierarchy of Figure2

The Object Base

Figure 4 presents a database fragment describing employees, students, and other entities. The first statement there says that the object *bob* has an attribute, *name*, that takes the value "*Bob*" on *bob*. Here *bob* is a logical id of an object that supposedly represents a person. (In fact, it represents a *faculty*, according to Figure 2.) In contrast, "*Bob*" is a member of class *string*; it represents the value of one of *bob*'s attributes, called *name*.

Note that F-logic does not emphasize the dichotomy between "objects" and "values." We believe that complications introduced by this dichotomy do not justify the benefits. Thus, in F-logic, "*Bob*" is viewed as an oid that represents itself, the string "*Bob*". However, the dichotomy between object ids and atomic values (*e.g.*, integers, strings, etc.) is easily represented in a sorted version of F-logic (Section 17.1). In addition, in Section 16.3 we discuss modeling of so called "complex values" — structures that are similar to complex objects, but which have no oid's.

Statement (i) also says that *bob* works in the department denoted via the oid cs_1, the department's name is represented by the oid "CS", and its manager is described by the object with oid *bob*. Note that *bob* has cyclic references to itself. Statement (ii) represents similar information about *mary*. Unlike the attributes *name*, *highestDegree*, and *age*, which return a single value, the attribute *friends* is *set-valued*. Syntactically, this is indicated by the double-headed arrow "$\twoheadrightarrow$" and the braces "{ }".

Statements (iii – vi) provide general information about classes and their *signatures*. A signature of a class specifies names of attributes and the methods that are applicable to this class, the type of arguments each method takes, and the type of the result it returns. Statement (iii), for instance, says that for every object in class *faculty*, the attributes *age* and *highestDegree* must be of types *midaged* and *degree*, respectively, and that the attribute *boss* returns results that simultaneously belong to classes *faculty* and *manager*.

In object-oriented systems, a *method* is a function of the form $Objects \times Objects^n \longrightarrow Objects$ or $Objects \times Objects^n \longrightarrow \mathcal{P}(Objects)$, where $\mathcal{P}$ is a power-set operator. Given an object, its methods are "encapsulated" inside that object and constitute its interface to the rest of the system. The first argument of a method is the object the method is invoked on—the *host-object* of the invocation. The other arguments are called *proper* arguments of the invocation.

The reader may have already noticed that double-headed arrows, $\twoheadrightarrow$ and $\Rrightarrow$, are used in conjunction with set-valued attributes, while $\rightarrow$ and $\Rightarrow$ signify scalar attributes. Also, double-*shafted* arrows, $\Rightarrow$ and $\Rrightarrow$, specify types, while the arrows $\rightarrow$ and $\twoheadrightarrow$ describe values of attributes. A double-shafted arrow

Database facts:

(i) $bob\,[\,name \rightarrow \text{“}Bob\text{”};$ % Defining a scalar property *name* of *bob*
$age \rightarrow 40;$ % Defining a scalar property *age* of *bob*
$affiliation \rightarrow cs_1[\,dname \rightarrow \text{“}CS\text{”};$
$mngr \rightarrow bob;$
$assistants \twoheadrightarrow \{john,\ sally\}]\,]$
(ii) $mary\,[\,name \rightarrow \text{“}Mary\text{”};$
$highestDegree \rightarrow ms;$
$friends \twoheadrightarrow \{bob,\ sally\};$ % Defining a set-valued property of *mary*
$affiliation \rightarrow cs_2[dname \rightarrow \text{“}CS\text{”}\,]\,]$

General class information:

(iii) $faculty[\,boss \Rightarrow (faculty, manager);$ % Typing: Scalar attribute *boss* returns objects belonging
$age \Rightarrow midaged;$ % simultaneously to classes *faculty* and *manager*.
$highestDegree \Rightarrow degree;$ % Attribute returning objects from class *degree*.
$papers \Rrightarrow article;$ % Returns *sets* of objects from class *article*.
$highestDegree \bullet\!\!\rightarrow phd;$ % Defining an inheritable property *highestDegree*.
$avgSalary \rightarrow 50000\,]$ % Defining a non-inheritable property *avgSalary*.
(iv) $person[\,name \Rightarrow string;$
$friends \Rrightarrow person;$ % Typing: Attribute *friends* returns sets of persons.
$children \Rrightarrow child(person)]$ % Returns sets of elements from class *child*(*person*).
(v) $empl\,[\,affiliation \Rightarrow dept;$
$boss \Rightarrow empl;$
$jointWorks\,@\,empl \Rrightarrow report\,]$ % Typing: Method *jointWorks* takes one argument of class
% *empl* and returns a set of *report*-objects.
(vi) $dept\,[\,assistants \Rrightarrow (student, empl);$ % Returns sets of objects that simultaneously
$mngr \Rightarrow empl\,]$ % belong to classes *student* and *empl*.

Deductive rules:

(vii) $E\,[boss \rightarrow M\,] \longleftarrow\ E : empl \wedge D : dept$
$\wedge\, E\,[affiliation \rightarrow D\,[mngr \rightarrow M : empl\,]\,]$
(viii) $X\,[jointWorks\,@\,Y \twoheadrightarrow Z\,] \longleftarrow\ Y : faculty \wedge X : faculty$
$\wedge\, Y\,[papers \twoheadrightarrow Z\,] \wedge X\,[papers \twoheadrightarrow Z\,]$

Queries:

(ix) $?-\ X : empl \wedge X\,[\,boss \rightarrow Y;$
$age \rightarrow Z : midaged;$
$affiliation \rightarrow D\,[dname \rightarrow \text{“}CS\text{”}\,]\,]$
(x)] $?-\ mary\,[jointWorks\,@\,Y \twoheadrightarrow jacm90\,]$
(xi)] $?-\ mary\,[jointWorks\,@\,phil \twoheadrightarrow Z\,]$

Figure 4: A Sample Database

specifies that an attribute (or a method) is defined; if it is not given, the single-shafted arrow may not be used with this attribute. For instance, given $cl\,[attr \Rightarrow \cdots]$, one cannot write $obj\,[attr \twoheadrightarrow \cdots]$, where obj is in class cl, to specify the value of $attr$, unless $[\,attr \Rightarrow\!\!\!\!\Rightarrow \cdots]$ is specified for this or other class of obj. Also, note that (a, b, c) in $cl\,[attr \Rightarrow\ (a, b, c)]$ signifies a conjunction of types: the value of $attr$ on any object in class cl must simultaneously belong to each of the classes a, b, and c. When the set of types to the right of a double-shafted arrow, "$\Rightarrow$" or "$\Rightarrow\!\!\!\!\Rightarrow$", is a singleton set, *e.g.*, $c\,[attr \Rightarrow\ (empl)]$, we shall omit the surrounding parentheses and write $c\,[attr \Rightarrow empl]$.

Different kinds of information about objects can be mixed in one statement. For instance, in Statement (iii), the expression $highestDegree \Rightarrow degree$ specifies the type of the attribute $highestDegree$; the expression $highestDegree \bullet\!\!\!\to phd$ specifies an *inheritable property* of *faculty*; and $avgSalary \rightarrow 50000$ is a *non-inheritable property* of *faculty*.

Asserting an inheritable property for a class-object has the effect that every member-object of that class inherits this property, unless it is overwritten. For instance, the assertion $bob\,[highestDegree \rightarrow phd]$ is derivable by inheritance from (iii) and from the fact $bob : faculty$. Note that when a property is inherited by a member of the class, it becomes *non-inheritable*; this explains why inheritance derives $bob\,[highestDegree \rightarrow phd]$ rather than $bob\,[highestDegree \bullet\!\!\!\to phd]$. An inheritable property may also be inherited by a subclass. However, in this case, the inherited property remains inheritable in this subclass and, as such, it can be passed further down in the hierarchy of objects. For example, if *lecturer* were a subclass of *faculty* then $lecturer\,[highestDegree \bullet\!\!\!\to phd]$ would be derivable by inheritance, unless *lecturer* had another inheritable property (*e.g.*, $highestDegree \bullet\!\!\!\to ms$) to override this inheritance. Inheritance will be discussed in detail in Section 15.

In contrast to inheritable properties, the property $avgSalary \rightarrow 50000$ in (iii) is not inheritable by the members and subclasses of *faculty*. And, indeed, it makes no sense to inherit average salary, as it is an aggregate property of all members of the class and has no meaning for individual members. Inheriting $avgSalary \rightarrow 50000$ to a subclass is also meaningless, because subclass members, *e.g.*, all lecturers, are likely to have a different average salary than all members of the larger class *faculty*.

Observe that the same property may be inheritable and non-inheritable at the same time, depending on the context. This duality may even take place within one object! For instance, let *format* be an attribute that specifies the space needed to represent an object. Then $integer\,[format \rightarrow byte;\ format \bullet\!\!\!\to word]$ asserts that representing the object *integer* itself takes one byte, while *representing* any member of class *integer* (such as 7182) takes one machine word. In other words, $format \rightarrow byte$ is a non-inheritable property of *integer* as an object, while $format \bullet\!\!\!\to word$ is an inheritable property of *integer* as a class of objects. Each individual object in that class, such as 7182, would inherit $format \rightarrow word$ as a non-inheritable property. To further confuse the reader, we could assert $datatype\,[format \bullet\!\!\!\to byte]$. In this case, since presumably $integer : datatype$ holds, $integer\,[format \rightarrow byte]$ would become derivable by inheritance and so this property will not need to be stated explicitly.

Statements (iv) – (vi) specify typing constraints for classes *person*, *empl*, and *dept*. More precisely, it can be said that these statements define *signatures* of methods attached to these classes. Two things are worth noting here. First, in (v), the expression $jointWorks\,@\,empl \Rightarrow\!\!\!\!\Rightarrow report$ describes a method, *jointWorks*, that expects one proper argument from class *empl* and returns a set of elements, each in class *report*. In object-oriented terms, this means that whenever a *person*-object, *obj*, receives a message, *jointWorks*, with an argument from class *empl*, then the reply-message returned by *obj* will consist of a set of objects, each being a member of class *report*. Note that there is no essential difference between methods and attributes: the latter are simply methods without arguments. Strictly speaking, in Figure 4 we should have written $name\,@\ \Rightarrow string$ and $age\,@\ \rightarrow 40$ instead of $name \Rightarrow string$ and $age \rightarrow 40$, but the latter short-hand notation is more convenient when no arguments are expected.

The second thing to note is the expression $children \Rightarrow\!\!\!\!\Rightarrow child(person)$ in (iv), which specifies a type constraint for the attribute *children*. Here, *child* is a unary function symbol and *person* is a constant denoting a class. The term $child(person)$, then, is a logical id of another class. Thus, in F-logic, function symbols are used as constructors of object and class id's.

We shall often omit braces surrounding singleton sets and write, say, $friends \twoheadrightarrow bob$ instead of $friends \twoheadrightarrow \{bob\}$. However, in $friends \twoheadrightarrow \{bob, sally\}$ in Statement (ii) and in $assistants \twoheadrightarrow \{john, sally\}$ in (i), braces must be kept to indicate sets. (Actually, $cs_1[assistants \twoheadrightarrow \{john, sally\}]$ is equivalent to a conjunction of $cs_1[assistants \twoheadrightarrow john]$ and $cs_1[assistants \twoheadrightarrow sally]$, but the use of braces leads to a more concise notation).

Statement (vii) is a deductive rule that defines a new attribute, *boss*, for objects of class *empl*. It says that an employee's boss is the manager of the department the employee works in. Here we follow the standard convention in logic programming that requires names of logical variables to begin with a capital letter. Statement (viii) defines a method, *jointWorks*, whose signature is given in (v). For any object in class *person*, *jointWorks* is a function that takes an argument of type *person* and returns a set of objects of type *report*; each object in this set represents a paper co-authored by the two people. Informally, this rule should be read as follows: If a *report*-object, Z, simultaneously belongs to the sets $X.papers$ and $Y.papers$,[1] where X and Y are *faculty*-objects, then Z must also belong to the set returned by the method *jointWorks* when invoked on the host-object X with the argument Y.

We emphasize that the variable Z ranges over the *members* of the set $X.papers \cap Y.papers$—it is *not* instantiated to the set itself. Note also that we do not need the restriction $Z : report$ in the rule body because, for each object in *faculty*, the attribute *papers* specifies a set of *article*-objects (by (iii)), and *article* is a subclass of *report* (by Figure 2). However, if we were interested in JACM papers only, then a restriction, such as $Z : jacm$, would have been necessary in the body of (viii).

Two uses of the method *jointWorks* are shown in (x) and (xi). Statement (x) is a query about co-authors of *mary*'s paper represented by the oid *jacm*90; the query (xi) requests all joint papers that *mary* co-authored with *phil*. Statement (ix) is a query about all middle-aged employees working for "*CS*" departments. In particular, for every such employee, the attributes *boss*, *age*, and *affiliation* are requested. The expected answer to (ix) is:

(xii) $bob\,[boss \rightarrow bob;\ age \rightarrow 40;\ affiliation \rightarrow cs_1]$.

The object *mary* does not qualify as an answer to (ix) because of the unknown age.

In this context, we would like to mention the fact that attributes and methods are *partial* functions. They may be defined on some objects in a class and undefined on another. F-logic distinguishes two reasons for undefinedness: 1) the attribute (or method) may be *inapplicable* to the object; or 2) it may be applicable but its value is unknown. Case 1 arises due to typing, which is discussed later. Case 2, on the other hand, is essentially a form of a *null value* known from the database theory.

In the above example and in the rest of this paper, the term "type" of an object is used interchangeably to refer to classes of that object and to its declared signatures. The exact meaning will be clear from the context. The dual use of this term is appropriate since, generally, the notion of type refers to arbitrary collections of abstract values. Classes and signatures both specify such collections. The former denotes

[1] $X.papers$ here denotes the set of oid's returned by the attribute *papers* on the object X, *i.e.*, the set of all Z such that $X\,[papers \twoheadrightarrow Z]$ holds. Similarly for $Y.papers$.

collections of class members, *i.e.*, of semantically related objects; the latter denotes collections of objects that are structurally related. Since semantic similarity usually implies structural similarity, the two uses of the term "type" are closely related.

Other Features

Before bidding farewell to our introductory example, we would like to highlight some other features of F-logic. Suppose that in Statement (i) we replace $mngr \rightarrow bob$ with $mngr \rightarrow phil$. Then we would have had a *type error*. Indeed, on the one hand, the deductive rule (vii) implies $bob[boss \rightarrow phil]$; on the other hand, from Figure 2, *phil* is neither a faculty nor a manager. This violates the typing constraint on the attribute *boss* in (iii).[2] As will be seen in Section 13, the notions of well-typing and type error have precise *model-theoretic* meaning in F-logic.

A related aspect of the type system of F-logic is that signature declarations are enforced. For instance, the following methods are declared in Figure 4 for the members of class *faculty*:

- *name*, *friends*, and *children*—the methods inherited from *person*;
- *affiliation* and *jointWorks*—methods inherited from *empl*; and
- *boss*, *age*, *highestDegree*, and *papers*—the methods directly specified for the class *faculty*.

Enforcing signature declarations means that these are the *only* methods applicable to the members of class *faculty*; any other method is illegal in the scope of that class. Moreover, it is a type error to invoke the method *jointWorks* with an argument that is not a member of class *empl*. Similarly, it would be a type error to invoke the methods in the last group on objects that represent employees who are not faculty, or to invoke *affiliation* or *jointWorks* on objects that are not members of class *empl*.

In our example, invocations of all methods, except *avgSalary*, are sanctioned by signatures declared for appropriate classes. The method *avgSalary*, on the other hand, is not covered by any signature, which is a violation of the well-typing conditions (to be discussed in Section 13). To correct the problem, the class *faculty* (in its role as an object) has to be made into a member of another class, say *employmentGroup* (whose members would be the various categories of employees, such as *empl*, *faculty*, and *manager*). To give *avgSalary* a proper type, we could then declare $employmentGroup[avgSalary \Rightarrow integer]$.

Observe a difference in the treatment of the non-inheritable method *avgSalary* (and of all other non-inheritable methods in the example) and of the inheritable method *highestDegree*. The former must be covered by a signature declared for a class where *faculty* is a *member*, while the latter should be covered by a signature declared for a class where *faculty* is a *subclass*.[3] This is because the non-inheritable method *avgSalary* is a property of the object *faculty* itself, while the inheritable method *highestDegree* is, effectively, a property of the objects that are members of the class *faculty*.

Yet another important aspect of F-logic type system is *polymorphism*. This means that methods can be invoked on different classes with arguments of different types. For instance, the following signatures

$$integer[plus@integer \Rightarrow integer] \qquad real[plus@real \Rightarrow real] \tag{1}$$

[2]Note that *phil* is an employee and, thus, $bob[boss \rightarrow phil]$ complies with the typing of *boss* for the class *empl*, as specified in (v). However, *bob* is also a member of class *faculty*, and (iii) imposes a stricter type on the attribute *boss* in that class, requiring *phil* to be a member of both *faculty* and *manager*.

[3]This class could be *faculty* itself, since the subclass-relationship in F-logic is non-strict.

say that *plus* is a method that returns an integer when invoked on an *integer*-object with an *integer*-argument. However, when *plus* is invoked on an object in class *real* with an argument that also comes from that class, the method returns an object in class *real*. In general, just as in (1), specifying polymorphic types requires more than one signature. The polymorphism of the kind illustrated in (1) is called *overloading*. Other examples of polymorphic types (including parameterized types) will be given in Section 12.3.

A form of polymorphism when a method can be invoked with varying number of arguments is called *arity-polymorphism*. For instance, $student[avgGrade \Rightarrow grade]$ and $student[avgGrade\,@year \Rightarrow grade]$ may both be legal invocations (say, the first implicitly assuming the current year).

Arity-polymorphism is very popular in logic programming languages, such as Prolog. However, unlike Prolog, arity polymorphism in F-logic is controlled via signatures, just as any other kind of polymorphism. This means that to be able to invoke a method with any given number of arguments, an appropriate signature must be specified. For instance, if no other signature is given, *avgGrade* cannot be invoked with more than one argument. Another way to control arity-polymorphism is by turning F-logic into a sorted language. This extension is described in Section 17.

One more kind of polymorphism arises when a method is declared to be both set-valued and scalar. For instance, suppose the following types are defined:

$$student[grade\,@course \Rightarrow\!\!\!\!\Rightarrow integer] \qquad\qquad student[grade\,@course \Rightarrow integer]$$

In the first case, *grade* is a set-valued method of one argument that for any student and any given course returns the set of this student's scores in the course (say, the scores for all projects and examinations). In the second case, only the overall grade for the course is returned. For instance, if *sally* is a student, the query

$$?-\quad sally[grade\,@db \twoheadrightarrow X]$$

will return the set of Sally's scores in the database course, while the query

$$?-\quad sally[grade\,@db \rightarrow X]$$

will return the final grade. This kind of polymorphism is controlled, as before, via signatures; it can also be controlled via sorts, similarly to arity-polymorphism.

Observe that F-logic manipulates several higher-order concepts. In Figure 4, the attribute *friends* is a set-valued function. Similarly, we can view the classes in the IS-A hierarchy of Figure 2 as sets ordered by the subset relation. Furthermore, attributes and methods are also viewed as objects. This, for instance, implies that their names can be returned as query answers. In this way, schema information is turned into data and it can be manipulated in the same language. This is useful for tasks that require schema exploration in databases (Section 12.4.2), inheritance with overriding (Section 15.3), and for many other applications (Sections 12.4.3 and 12.4.5).

Despite the higher-order syntax, the underlying semantics of F-logic formally remains first-order,[4] which is how we circumvented the difficulties normally associated with higher-order theories. These important issues are beyond the scope of this paper; a more complete discussion appears in [34].

[4]Following [34], being first-order here means that variables do not range over complex domains, such as the domain of sets or the domain of functions, but they can range over the intensions of those higher-order entities.

4 Syntax

The *alphabet* of an F-logic language, $\mathcal{L}$, consists of:

- a set of *object constructors*, $\mathcal{F}$;
- an infinite set of *variables*, $\mathcal{V}$;
- auxiliary symbols, such as, (,), [,], $\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\rightarrow$, $\bullet\!\!\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!\!\!\Rightarrow$, etc; and
- usual logical connectives and quantifiers, $\vee$, $\wedge$, $\neg$, $\longleftarrow$, $\forall$, $\exists$.

Object constructors (the elements of $\mathcal{F}$) play the role of function symbols of F-logic. Each function symbol has an *arity*—a nonnegative integer that determines the number of arguments this symbol can take. Symbols of arity 0 are also called *constants*; symbols of arity ≥ 1 are used to construct larger terms out of simpler ones. An *id-term* is a usual first-order term composed of function symbols and variables, as in predicate calculus. The set of all *ground* (*i.e.*, variable-free) id-terms is denoted by $U(\mathcal{F})$. This set is also commonly known as *Herbrand Universe*.

Conceptually, ground id-terms play the role of *logical object id's*—a logical abstraction of the implementational concept of *physical object identity* [53, 3]. Since this paper is about a logic, the term "object id" (abbr., *oid*) will be used for logical id's only.

Objects represented by "compound" id-terms, such as $addr(13, mainstreet, anywhere)$, usually arise when a complex object (or a class) is constructed out of simpler components, *e.g.*, 13, *mainstreet*, and *anywhere*, in this example.

Throughout this paper, $\mathcal{F}$ and $U(\mathcal{F})$ will be used to denote the set of function symbols and ground terms, respectively, where the language, $\mathcal{L}$, will be known from the context.

Molecular Formulas

A language of F-logic (henceforth called *F-language*, for brevity) consists of a set of formulae constructed out of the alphabet symbols. As in many other logics, formulas are built out of simpler formulas by using the usual connectives $\neg$, $\vee$, and $\wedge$, and quantifiers $\exists$ and $\forall$. The simplest kind of formulas are called *molecular* F-formulas (abbr., *F-molecules* or just *molecules*).

We adopt a convention inspired by Prolog's syntax, whereby a symbol that begins with a lower-case letter denotes a ground id-term and a symbol that begins with a capital letter denotes an id-term that may be non-ground.

Definition 4.1 (Molecular Formulas) A *molecule* in F-logic is one of the following statements:

(i) An *is-a assertion* of the form $C :: D$ or of the form $O : C$, where C, D, and O are id-terms.

(ii) An *object molecule* of the form O[a ';'-separated list of *method expressions*].
A *method expression* can be either a *non-inheritable data expression*, an *inheritable data expression*, or a *signature expression*.

- *Non-inheritable data expressions* take one of the following two forms:

- A non-inheritable *scalar* expression ($k \geq 0$):

$$ScalarMethod\,@\,Q_1, \ldots, Q_k \rightarrow T$$

- A non-inheritable *set-valued* expression ($l, m \geq 0$):

$$SetMethod\,@\,R_1, \ldots, R_l \twoheadrightarrow \{S_1, \ldots, S_m\}$$

- *Inheritable* scalar and set-valued data expressions are like non-inheritable expressions except that "$\rightarrow$" is replaced with "$\bullet\!\!\rightarrow$" and "$\twoheadrightarrow$" is replaced with $\bullet\!\!\twoheadrightarrow$.
- *Signature expressions* also take two forms:
  - A *scalar* signature expression ($n, r \geq 0$):

$$ScalarMethod\,@\,V_1, \ldots, V_n \Rightarrow (A_1, \ldots, A_r)$$

  - A *set-valued* signature expression ($s, t \geq 0$):

$$SetMethod\,@\,W_1, \ldots, W_s \Rrightarrow (B_1, \ldots, B_t)$$ □

The first is-a assertion in (i), $C :: D$, states that C is a *nonstrict* subclass of D (*i.e.*, inclusive of the case when C and D denote the same class).[5] The second assertion, $O : C$, states that O is a member of class C.

In (ii), O is an id-term that denotes an object. *ScalarMethod* and *SetMethod* are also id-terms. However, the syntactic context of *ScalarMethod* indicates that it is invoked on O as a scalar method, while the context of *SetMethod* indicates a set-valued invocation. (If *ScalarMethod* and *SetMethod* are terms that have variables in them, then each term represents a family of methods rather than a single method.) Double-headed arrows, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, and $\Rrightarrow$, indicate that *SetMethod* denotes a set-valued function. The single-headed arrows, $\rightarrow$, $\bullet\!\!\rightarrow$, and $\Rightarrow$, indicate that the corresponding method is scalar.

In the above data expressions, T and S_i are id-terms that represent output of the respective methods, *ScalarMethod* and *SetMethod*, when they are invoked on the host-object O with the arguments $Q_1, \ldots,$ Q_k and $R_1, \ldots, R_l$, respectively. The arguments are id-terms.

In the signature expressions, A_i and B_j are id-terms that represent *types* of the results returned by the respective methods when they are invoked on an object of class C with arguments of types $V_1, \ldots,$ V_n and $W_1, \ldots, W_s$, respectively; these arguments are also id-terms. The notation "$(\cdots)$" in signature expressions is intended to say that the output of the method must belong to *all* the classes listed in parentheses to the right of "$\Rightarrow$" or "$\Rrightarrow$".

The order of data and signature expressions in a molecule is immaterial. For convenience, the same method and even the same data/signature expression may have multiple occurrences in the same molecule. Likewise, the same id-term may occur multiple times in the braces inside a data expression or in the parentheses inside a signature expression. Furthermore, whenever a method does not expect arguments, "@" will be omitted. For instance, we will write $P[Mthd \rightarrow Val]$ instead of $P[Mthd\,@ \;\rightarrow Val]$, and similarly for $\twoheadrightarrow$, $\Rightarrow$, and $\Rrightarrow$. Likewise, when only one element appears inside { }, we may write $P[\ldots \twoheadrightarrow S]$ instead of $P[\ldots \twoheadrightarrow \{S\}]$; similarly, we shall write $Q[\ldots \Rightarrow T]$ and $Q[\ldots \Rrightarrow T]$ instead of $Q[\ldots \Rightarrow (T)]$ and $Q[\ldots \Rrightarrow (T)]$.

[5]Assertions, such as $person :: person$, will later turn out to be tautologies, *i.e.*, any class is a subclass of itself.

Discussion

Informally, an object molecule in (ii) above asserts that the object denoted by O has properties specified by the method expressions listed inside the brackets "$[\ldots]$". Data expressions are used to define properties of objects in terms of what their methods are supposed to do. Inheritable data expressions may be inherited by subclasses and individual members of the object (when it plays the role of a class); in contrast, properties specified as non-inheritable cannot be inherited. A signature expression in (ii) specifies type constraints on the methods applicable to the objects in class O. Typing is given both for method arguments and for their results. For instance, the scalar signature expression in (ii) says that $ScalarMethod$ is a scalar method (indicated with a "$\Rightarrow$") and that when it is invoked on a host-object of class O with proper arguments coming from classes $V_1, \ldots, V_n$, then the result must simultaneously belong to classes $A_1, \ldots, A_r$. Similarly, the typing for $SetMethod$ in (ii) says that it is a set-valued method (indicated with a "$\Rightarrow\!\!\!\!\Rightarrow$"); when it is invoked on a host-object of class O with proper arguments from classes $W_1, \ldots, W_s$, then *each* element in the resulting set must simultaneously belong to classes $B_1, \ldots, B_t$.

Notice that a molecule, such as $a\,[attr \rightarrow b; attr \twoheadrightarrow \{c, d\}; attr \bullet\!\!\rightarrow e]$, is syntactically well-formed despite the fact that $attr$ is used to specify a non-inheritable scalar property of the object a in one data expression, a non-inheritable set-valued property in another, and also an inheritable scalar property in the third part. This apparent contradiction is easily resolved at the semantic level: The attribute $attr$ has value b on object a when invoked as a non-inheritable scalar method (with the arrow "$\rightarrow$"); it returns the set $\{c, d\}$ when called as a non-inheritable set-valued method (with "$\twoheadrightarrow$"); and it returns the value e when called as an inheritable scalar method (with the arrow "$\bullet\!\!\rightarrow$"). If a happens to be a class-object, the properties $attr \rightarrow b$ and $attr \twoheadrightarrow \{c, d\}$ are not inheritable by the members and subclasses of a. However, $attr \bullet\!\!\rightarrow e$ is inheritable, since it is specified as such.

It follows from the syntax that every logical id can denote an entity or a method, depending on the syntactic context of this id within the formula. If it occurs as a method, this id can denote either a *scalar* function or a *set-valued* function. The type of the invocation (scalar or set-valued) is determined by the context, *viz.*, by the type of the associated arrow.

Is-a assertions are always atomic—they cannot be decomposed further into simpler formulas. Other molecular formulas, however, are not always atomic. As will be seen shortly, a molecule such as $X\,[attr_1 \rightarrow a; attr_2 \rightarrow Y]$ is equivalent to a conjunction of its *atoms*, $X\,[attr_1 \rightarrow a] \wedge X\,[attr_2 \rightarrow Y]$. It is for this property that we call such formulas "molecular" instead of "atomic." Atomic formulas will be defined later, in Section 7.

Complex Formulas

F-formulae are built out of simpler F-formulae by means of logical connectives and quantifiers:

- Molecular formulae are F-formulas;
- $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$ are F-formulae, if so are φ and ψ;
- $\forall X\, \varphi$, $\exists Y\, \psi$ are formulae, if so are φ, ψ and X, Y are variables.

In addition, we define a *literal* to be either a molecular formula or a negation of a molecular formula.

In Section 3 and elsewhere in this paper, we shall often use the *implication* connective, " $\longleftarrow$ ". In F-logic, this connective is defined as in classical logic: $\varphi \longleftarrow \psi$ is a shorthand for $\varphi \vee \neg\psi$. There is a tradition to refer to logical statements written as implications as *rules*. This terminology was already used in the example of Section 3, and we shall continue this practice.

It is sometimes convenient to combine different kinds of molecules (as in (vii), (viii) and (ix) of Figure 4) and write, say,

$$Q : P\,[ScalM \,@\, X \rightarrow (Y : S);\ SetM \,@\, Y, W \Rightarrow\!\!\!\!\Rightarrow (Z : R, T)]$$

as an abbreviation for

$$Q : P \wedge Q\,[ScalM \,@\, X \rightarrow Y] \wedge Y : S \wedge Q\,[SetM \,@\, Y, W \Rightarrow\!\!\!\!\Rightarrow (Z, T)] \wedge Z : R.$$

Furthermore, even though the symbols on the right-hand side of the arrows denote id-terms by definition, it is often convenient (and, in fact, customary) to combine molecules as in (i) and (ii) of Figure 4. For instance,

$$P\,[ScalM \,@\, X \rightarrow Q\,[SetM \,@\, Y \twoheadrightarrow \{T, S\}]]$$

can be defined as an abbreviation for

$$P\,[ScalM \,@\, X \rightarrow Q] \wedge Q\,[SetM \,@\, Y \twoheadrightarrow \{T, S\}]$$

5 Semantics

Given a pair of sets, U and V, we shall use $Total(U, V)$ to denote the set of all total functions $U \longrightarrow V$; similarly, $Partial(U, V)$ stands for the set of all partial functions $U \longrightarrow V$. The power-set of U will be denoted by $\mathcal{P}(U)$. Further, given a collection of sets $\{S_i\}_{i \in \mathcal{N}}$ parameterized by natural numbers, $\prod_{i=1}^{\infty} S_i$ will denote the Cartesian product of the S_i's, that is, the set of all infinite tuples $\langle s_1, \ldots, s_n, \ldots \rangle$.

5.1 F-structures

In F-logic, semantic structures are called *F-structures*. Given an F-language, $\mathcal{L}$, an *F-structure* is a tuple $\mathbf{I} = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\rightarrow}, I_{\bullet\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\!\Rightarrow} \rangle$. Here U is the domain of $\mathbf{I}$, $\prec_U$ is an irreflexive partial order on U, and $\in_U$ is a binary relation. As usual, we write $a \preceq_U b$ when $a \prec_U b$ or $a = b$. We extend $\preceq_U$ and $\in_U$ to tuples over U in a natural way: for any $\vec{u}, \vec{v} \in U^n$, we write $\vec{u} \preceq_U \vec{v}$ or $\vec{u} \in_U \vec{v}$ if the corresponding relationships hold between $\vec{u}$ and $\vec{v}$ component-wise.

The ordering $\prec_U$ on U is a semantic counterpart of the subclass-relationship, *i.e.*, $a \prec_U b$ is interpreted as a statement that a is a subclass of b. The binary relation $\in_U$ will be used to model class membership, *i.e.*, $a \in_U b$ should be taken to mean that a is a member of class b. The two binary relationships, $\prec_U$ and $\in_U$, are related as follows: If $a \in_U b$ and $b \preceq_U c$ then $a \in_U c$. This is just another way of saying that the *extension* of a subclass (*i.e.*, its set of members) is a subset of the extension of a superclass.

We do not impose any other restrictions on the class membership relation, which lets us accommodate a wide range of applications. In particular, $\in_U$ does not have to be acyclic and even $s \in_U s$ is a possibility (*i.e.*, a class may be a member of itself when viewed as an object).[6] The reader should not be misled

[6]Such flexibility is sometimes required in AI applications, where s would be interpreted as a "typical" element of class s.

into thinking that v in $u \in_U v$ is a subset of U that contains u. The actual meaning of such a statement is that v is an element of U that *denotes* a subset of U that contains u.

By analogy with classical logic, we can view U as a set of all *actual* objects in a *possible world*, **I**. Ground id-terms (the elements of $U(\mathcal{F})$) play the role of logical object id's. They are interpreted by the objects in U via the mapping $I_{\mathcal{F}} : \mathcal{F} \longrightarrow \cup_{i=0}^{\infty} Total(U^i, U)$. This mapping interprets each k-ary object constructor, $f \in \mathcal{F}$, by a function $U^k \longrightarrow U$. For $k = 0$, $I_{\mathcal{F}}(f)$ can be identified with an element of U. Thus, function symbols are interpreted the same way as in predicate calculus.

The remaining six symbols in **I** denote mappings for interpreting each of the six types of method expressions in F-logic. These mappings are described next.

5.1.1 Attaching Functions to Methods — The Mappings $I_{\rightarrow}$, $I_{\twoheadrightarrow}$, $I_{\bullet\!\rightarrow}$, and $I_{\bullet\!\twoheadrightarrow}$

As in classical logic, the mapping $I_{\mathcal{F}}$ above is used to associate an element of U with each id-term. However, in F-logic, id-terms can also be used to denote methods. A method is a function that takes a host-object and a list of proper arguments and maps them into another object or a set of objects, depending on whether the method is invoked as a scalar or a set-valued function. Therefore, to assign meaning to methods, an F-structure has to attach an appropriate function to each method.

Since id-terms can play the role of method names, each id-term will have a function for representing the actual method named by this term. There is one subtlety, though. Using id-terms as method names is not particularly useful, unless variables over method names are also allowed, which would make querying the database schema possible. It turns out that to give a meaning to molecules with method-variables, it is necessary to associate functions with the elements of U, rather than with the elements of $U(\mathcal{F})$.[7] Furthermore, since any method name can be used with different arities, an id-term needs a separate function for *each* possible arity.

Formally, in their role as methods, objects are interpreted via the assignment of appropriate functions to each element of U, using the maps $I_{\rightarrow}$, $I_{\bullet\!\rightarrow}$, $I_{\twoheadrightarrow}$, and $I_{\bullet\!\twoheadrightarrow}$. Specifically, for every object, its incarnation as a scalar method is obtained via the mappings:

- $I_{\rightarrow}, I_{\bullet\!\rightarrow} \; : U \longrightarrow \prod_{k=0}^{\infty} Partial(U^{k+1}, U)$

Each of these mappings associates a tuple of partial functions $\{f_k \; : \; U^{k+1} \rightarrow U \mid k \geq 0\}$ with every element of U; there is exactly one such f_k in the tuple, for every method-arity $k \geq 0$. In other words, the same method can be invoked with different arities.

In addition to different arities, every method can be invoked as a scalar or as a set-valued function (e.g. *sally* $[$*grade* @ *db*$\twoheadrightarrow X]$ and *sally* $[$*grade* @ *db* $\rightarrow X]$ in Section 3). Semantically this is achieved by interpreting the set-valued incarnations of methods via a different pair of mappings:

- $I_{\twoheadrightarrow}, I_{\bullet\!\twoheadrightarrow} \; : U \longrightarrow \prod_{k=0}^{\infty} Partial(U^{k+1}, \mathcal{P}(U))$

For every method-arity k, each of these mappings associates a partial function $U^{k+1} \rightarrow \mathcal{P}(U)$ with each element of U. Note that each element of U has four different sets of interpretations: two provided by $I_{\rightarrow}$ and $I_{\bullet\!\rightarrow}$ and two provided by $I_{\twoheadrightarrow}$ and $I_{\bullet\!\twoheadrightarrow}$.

[7]This is a subtle technical point whose necessity will become apparent once the notion of truth in F-structures is defined.

The difference between the "$\rightarrow$"-versions and the "$\bullet\!\!\rightarrow$"-versions in the above mappings is that the "$\rightarrow$"-versions are used to interpret non-inheritable data properties, while the "$\bullet\!\!\rightarrow$"-versions are for the inheritable ones. The distinction between inheritable and non-inheritable properties was explained in Section 3; the formal treatment will be given in Section 15.

As seen from the above definitions, $I_{\rightarrow}(m)$ (and $I_{\twoheadrightarrow}(m)$, $I_{\bullet\!\rightarrow}(m)$, $I_{\bullet\!\twoheadrightarrow}(m)$), where $m \in U$, is an infinite tuple of functions parameterized by the arity $k \geq 0$. To refer to the k-th component of such a tuple, we use the notation $I_{\rightarrow}^{(k)}(m)$ (resp., $I_{\twoheadrightarrow}^{(k)}(m)$, $I_{\bullet\!\rightarrow}^{(k)}(m)$, or $I_{\bullet\!\twoheadrightarrow}^{(k)}(m)$). Thus, a method, m, that occurs in a scalar non-inheritable data expression with k proper arguments is interpreted by $I_{\rightarrow}^{(k)}(m)$; if m occurs in a set-valued non-inheritable data expression with k arguments, it is interpreted by $I_{\twoheadrightarrow}^{(k)}(m)$, and so on. Note that $I_{\rightarrow}^{(k)}(m)$ and the other three mappings are $(k+1)$-ary functions. The first argument here is the host-object of the invocation of the method m; the other k arguments correspond to the *proper* arguments of the invocation. In the parlance of object-oriented systems, $I_{\rightarrow}^{(k)}(m)(obj, a_1, \ldots, a_k)$ is a request to object obj to invoke a scalar method, m, on the arguments $a_1, \ldots, a_k$.

5.1.2 Attaching Types to Methods — The Mappings $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$

Since methods are interpreted as functions, the meaning of a signature expressions must be a functional type, for its role is to specify the type of a method. To specify a functional type, one must describe the types of the arguments to which the function can be applied and the types of the results returned by the function. Furthermore, the description should account for polymorphic types (cf. (1) earlier).

Model-theoretically, functional types are described via the mappings $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$ that satisfy the usual properties of polymorphic functional types, as spelled out below. As in the case of $I_{\rightarrow}$ and related functions, these mappings are attached to the elements of U rather than $U(\mathcal{F})$.

We start with a rather succinct definition of $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$ and then continue with a discussion on the properties of these mappings:[8]

- $I_{\Rightarrow} : U \longrightarrow \prod_{i=0}^{\infty} PartialAntiMonotone_{\prec_U}(U^{i+1}, \mathcal{P}_{\uparrow}(U))$
- $I_{\Rightarrow\!\!\!\Rightarrow} : U \longrightarrow \prod_{i=0}^{\infty} PartialAntiMonotone_{\prec_U}(U^{i+1}, \mathcal{P}_{\uparrow}(U))$

Here $\mathcal{P}_{\uparrow}(U)$ is a set of all *upward-closed* subsets of U. A set $V \subseteq U$ is *upward closed* if $v \in V$ and $v \prec_U v'$ (where $v' \in U$) imply $v' \in V$. When V is viewed as a set of classes, upward closure simply means that, along with each class $v \in V$, this set also contains all the superclasses of v. $PartialAntiMonotone_{\prec_U}(U^{i+1}, \mathcal{P}_{\uparrow}(U))$ denotes the set of partial *anti-monotonic* functions from U^{i+1} to $\mathcal{P}_{\uparrow}(U)$. For a partial function $\rho : U^k \longrightarrow \mathcal{P}_{\uparrow}(U)$, *anti-monotonicity* means that if $\vec{u}, \vec{v} \in U^k$, $\vec{v} \preceq_U \vec{u}$, and $\rho(\vec{u})$ is defined, then $\rho(\vec{v})$ is also defined and $\rho(\vec{v}) \supseteq \rho(\vec{u})$.[9]

5.1.3 Discussion: The Relationship between $I_{\rightarrow}$ and $I_{\Rightarrow}$

The definition of F-structures is now complete. In the rest of this subsection, we discuss the properties of $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$ and show that they coincide with the standard properties of functional types (*e.g.*, [29]),

[8]Signatures constitute a fairly advanced level of F-logic; its *basic* features—is-a hierarchy and data expressions—do not depend on the specifics of the above type system. For this reason, on the first reading it is possible to skip signature-related aspects of F-logic, including the remaining part of this subsection.

[9]Actually, these functions are *monotone* with respect to *Smyth's ordering* [28]. For upward-closed sets, $S \subseteq^{smyth} S'$ if and only if $S \supseteq S'$.

albeit expressed in a different, model-theoretic notation. As with $I_{\rightarrow}$, we use $I^{(k)}_{\Rightarrow}(m)$ to refer to the k-th component of the tuple $I_{\Rightarrow}(m)$. We use similar notation, $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}(m)$, for set-valued methods.

The intended meaning of $I^{(k)}_{\Rightarrow}(m)$ is the type of the $(k+1)$-ary function $I^{(k)}_{\rightarrow}(m)$. In other words, the domain of definition of $I^{(k)}_{\Rightarrow}(m)$ is a set of $(k+1)$-tuples of classes, $\langle$*host-cl*, *arg-cl*$_1$, ... , *arg-cl*$_k\rangle$, that specify the type for the tuples of arguments, $\langle o, arg_1, \ldots, arg_k\rangle$, to which $I^{(k)}_{\rightarrow}(m)$ can be meaningfully applied (these are those argument-tuples for which $\langle o, arg_1, \ldots, arg_k\rangle \in_U \langle host\text{-}cl, arg\text{-}cl_1, \ldots, arg\text{-}cl_k\rangle$ holds). For every tuple of classes, $\langle host\text{-}cl, \overrightarrow{arg\text{-}cls}\rangle \in U^{k+1}$, if $I^{(k)}_{\Rightarrow}(m)(host\text{-}cl, \overrightarrow{arg\text{-}cls})$ is defined, it represents the type of $I^{(k)}_{\rightarrow}(m)(o, \overrightarrow{args})$ for any tuple of arguments such that $\langle o, \overrightarrow{args}\rangle \in_U \langle host\text{-}cl, \overrightarrow{arg\text{-}cls}\rangle$. This means that if $v = I^{(k)}_{\rightarrow}(m)(o, \overrightarrow{args})$ is defined, $v \in_U w$ must hold for *every* class, w, in $I^{(k)}_{\Rightarrow}(m)(host\text{-}cl, \overrightarrow{arg\text{-}cls})$. In particular, a set of types is interpreted essentially as an extended "and" of its members. This is the reason for the notation "$(\cdots)$" in the signatures (cf. Figure 4).

Similarly, the meaning of $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}(m)$ is defined to be the type of the set-valued function $I^{(k)}_{\twoheadrightarrow}(m)$. In this case, since $I^{(k)}_{\twoheadrightarrow}(m)(o, \overrightarrow{args})$ is a *set* of objects, *every* object in $v \in I^{(k)}_{\twoheadrightarrow}(m)(o, \overrightarrow{args})$ must belong to *every* class, w, in $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}(m)(host\text{-}cl, \overrightarrow{arg\text{-}cls})$, *i.e.*, the relationship $v \in_U w$ must hold.

The rationale behind the conditions of anti-monotonicity and upward-closure in the definition of $I^{(k)}_{\Rightarrow}(m)$ and $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}(m)$ should now become clear from the intended meaning of these functions. For instance, if the object $I^{(k)}_{\rightarrow}(m)(o, \overrightarrow{args})$ is of type cl (*i.e.*, $I^{(k)}_{\rightarrow}(m)(o, \overrightarrow{args}) \in_U cl$) then, clearly, $I^{(k)}_{\rightarrow}(m)(o, \overrightarrow{args})$ must be a member of every superclass of cl (*i.e.*, $I^{(k)}_{\rightarrow}(m)(o, \overrightarrow{args}) \in_U cl'$, for every cl' such that $cl \prec_U cl'$); thus $I^{(k)}_{\Rightarrow}(m)(host\text{-}cl, \overrightarrow{arg\text{-}cls})$ must be upward-closed. Similarly, if $I^{(k)}_{\rightarrow}(m)$ can be invoked on every member of class *host-cl* with proper arguments of type $\overrightarrow{arg\text{-}cls}$, then m must also be invocable on every member, o, of any subclass *host-cl*$'$ of *host-cl* with arguments $\overrightarrow{args}$ such that $\overrightarrow{args} \in_U \overrightarrow{arg\text{-}cls}'$, where $\langle host\text{-}cl', \overrightarrow{arg\text{-}cls}'\rangle \preceq_U \langle host\text{-}cl, \overrightarrow{arg\text{-}cls}\rangle$. Furthermore, the result of this application, $v = I^{(k)}_{\rightarrow}(m)(\langle o, \overrightarrow{args}\rangle)$, must still be typed by $I^{(k)}_{\Rightarrow}(m)(host\text{-}cl, \overrightarrow{arg\text{-}cls})$. Now, since the type of v is given by $I^{(k)}_{\Rightarrow}(m)(host\text{-}cl', \overrightarrow{arg\text{-}cls}')$, we obtain $I^{(k)}_{\Rightarrow}(m)(host\text{-}cl', \overrightarrow{arg\text{-}cls}') \supseteq I^{(k)}_{\Rightarrow}(m)(host\text{-}cl, \overrightarrow{arg\text{-}cls})$, *viz.*, anti-monotonicity.

The above relationship between $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\Rightarrow}$ and $I_{\rightarrow}$, $I_{\twoheadrightarrow}$ is not part of the definition of F-structures. Instead, it is captured at the meta-level, by the definition of type-correctness in Section 13. The relationship between $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\Rightarrow}$ and $I_{\bullet\!\!\rightarrow}$, $I_{\bullet\!\!\twoheadrightarrow}$, is also captured by the notion of type-correctness. However, it is slightly different from the relationship between $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\Rightarrow}$ and $I_{\rightarrow}$, $I_{\twoheadrightarrow}$ described above; it is fully explained in Section 13.

5.2 Satisfaction of F-formulas by F-structures

A *variable assignment*, ν, is a mapping from the set of variables, $\mathcal{V}$, to the domain U. Variable assignments extend to id-terms in the usual way: $\nu(d) = I_{\mathcal{F}}(d)$ if $d \in \mathcal{F}$ has arity 0 and, recursively, $\nu(f(\ldots, T, \ldots)) = I_{\mathcal{F}}(f)(\ldots, \nu(T), \ldots)$.

Molecular Satisfaction

Let $\mathbf{I}$ be an F-structure and ν a variable assignment. Intuitively, a molecule $T[\cdots]$ is *true* under $\mathbf{I}$ with respect to a variable assignment ν, denoted $\mathbf{I} \models_{\nu} T[\cdots]$, if the object $\nu(T)$ in $\mathbf{I}$ has properties that the formula $T[\cdots]$ says it has. As a special case, molecules of the form $T[\,]$, which specify no properties, are vacuously true. An is-a molecule, $P :: Q$ or $P : Q$, is true if the objects involved, $\nu(P)$ and $\nu(Q)$, are related via $\prec_U$ or $\in_U$ to each other.

Definition 5.1 (Satisfaction of F-Molecules) Let $\mathbf{I}$ be an F-structure and G be an F-molecule. We write $\mathbf{I} \models_\nu G$ if and only if all of the following holds:

- When G is an is-a assertion then:

  (i) $\nu(Q) \preceq_U \nu(P)$, if $G = Q :: P$; or
  $\nu(Q) \in_U \nu(P)$, if $G = Q : P$.

- When G is an object molecule of the form O [*a ';'-separated list of method expressions*] then for every method expression E in G, the following conditions must hold:

  (ii) If E is a non-inheritable scalar data expression of the form $ScalM \,@\, Q_1, \ldots, Q_k \rightarrow T$, the element $I^{(k)}_{\rightarrow}(\,\nu(ScalM)\,)(\nu(O), \nu(Q_1), \ldots, \nu(Q_k))$ must be defined and equal $\nu(T)$.

  Similar conditions must hold if E is an inheritable scalar data expressions, except that $I^{(k)}_{\rightarrow}$ should be replaced with $I^{(k)}_{\bullet\!\rightarrow}$.

  (iii) If E is a non-inheritable set-valued data expression, of the form $SetM \,@\, R_1, \ldots, R_l \twoheadrightarrow \{S_1, \ldots, S_m\}$, the set $I^{(l)}_{\twoheadrightarrow}(\,\nu(SetM)\,)(\nu(O), \nu(R_1), \ldots, \nu(R_l))$ must be defined and *contain* the set $\{\nu(S_1), \ldots, \nu(S_m)\}$.

  Similar conditions must hold if E is an inheritable set-valued data expression, except that $I^{(k)}_{\twoheadrightarrow}$ should be replaced with $I^{(k)}_{\bullet\!\twoheadrightarrow}$.

  (iv) If E is a scalar signature expression, $ScalM \,@\, Q_1, \ldots, Q_n \Rightarrow (R_1, \ldots, R_u)$, then the set $I^{(n)}_{\Rightarrow}(\nu(ScalM))(\,\nu(O), \nu(Q_1), \ldots, \nu(Q_n)\,)$ must be defined and contain $\{\nu(R_1), \ldots, \nu(R_u)\}$.

  (v) If E is a set-valued signature expression of the form $SetM \,@\, V_1, \ldots, V_s \Rightarrow\!\!\!\Rightarrow (W_1, \ldots, W_v)$, the set $I^{(s)}_{\Rightarrow\!\!\!\Rightarrow}(\nu(SetM))(\,\nu(O), \nu(V_1), \ldots, \nu(V_s)\,)$ must be defined and contain $\{\nu(W_1), \ldots, \nu(W_v)\}$.

□

Here, (i) says that the object $\nu(Q)$ must be a subclass or a member of the class $\nu(P)$. Conditions (ii) and (iii) say that—in case of a data expression—the interpreting function must be defined on appropriate arguments and yield results compatible with those specified by the expression. Conditions (iv) and (v) say that, for a signature expression, the type of a method ($ScalM$ or $SetM$) specified by the expression must comply with the type assigned to this method by $\mathbf{I}$.

The following observations may help shed some light on the rationale behind some aspects of the definition of F-structures:

- It follows from (iv) and (v) above that a signature of the form $c\,[m@c_1, \ldots, c_k \Rightarrow (\,)]$ (with nothing inside the parentheses) is *not* a tautology. Indeed, there are F-structures in which $I^{(k)}_{\Rightarrow}(\nu(m))(\nu(c), \nu(c_1), \ldots, \nu(c_k))$ is undefined. Similarly for $\Rightarrow\!\!\!\Rightarrow$.

  Such an "empty" signature intuitively says that the respective scalar method is *applicable* to objects in class c with arguments drawn from classes $c_1, \ldots, c_k$, but it does not specify the actual type of the result. The method, m cannot be applied in the above context without the signature $c\,[m@c_1, \ldots, c_k \Rightarrow (\,)]$ being true. These and other constraints are part of the well-typing conditions discussed in Section 13.

- When $c\,[m@c_1, \ldots, c_k \Rightarrow (\,)]$ is true in **I** but $I_{\rightarrow}(\nu(m))(\nu(o), \nu(a_1), \ldots, \nu(a_k))$ is *undefined*, where $\nu(o) \in_U \nu(c)$ and $\nu(c_1) \in_U \nu(a_1), \ldots, \nu(c_k) \in_U \nu(a_k)$, we have a *null value* of the kind "value missing," a situation well-known in the database theory [103]. When $c\,[m@c_1, \ldots, c_k \Rightarrow (\,)]$ is false, the well-typing conditions of Section 13 mandate that $I_{\rightarrow}(\nu(m))(\nu(o), \nu(a_1), \ldots, \nu(a_k))$ *must* be undefined. This is analogous to a null value of the kind "value inapplicable," meaning that the value of the invocation of m on o with arguments $a_1, \ldots, a_n$ does not *and cannot* exist.

- The deceptive simplicity of items (iv) and (v) is a direct result of the upward-closure of the co-domain of $I_{\Rightarrow}(\nu(ScalM))$ and $I_{\Rightarrow\!\!\!\Rightarrow}(\nu(SetM))$. Without the upward-closure, (iv) would have looked like this:

  $I^{(n)}_{\Rightarrow}(\nu(ScalM))(\nu(O), \nu(Q_1), \ldots, \nu(Q_n))$ *must be defined and, for every* i *(*$1 \le i \le u$*), there must be an element,* p*, in this set such that* $p \preceq_U \nu(R_i)$.

  A similar change would have been needed for (v).

  A more serious consequence of dumping the upward-closure condition would be that the elegant anti-monotonicity constraint on $I_{\Rightarrow}(\nu(ScalM))$ and $I_{\Rightarrow\!\!\!\Rightarrow}(\nu(SetM))$ will have to be replaced with a rather awkward condition needed to coerce signatures into behaving as functional types (see Section 7.3).

- Notice *how* the above definition determines the meaning of molecules of the form $a\,[M\,@\,X \rightarrow b]$, $c\,[M \Rightarrow\!\!\!\Rightarrow d]$, etc. The subtlety here is in how method-functions are associated with M — a variable that ranges over methods.

  If ν is a variable assignment, then $\nu(M)$ is an element in U, not in $U(\mathcal{F})$. Therefore, associating method-functions with ground id-terms would have been useless for interpreting the above molecules. This explains why earlier we insisted that $I_{\rightarrow}$, $I_{\rightarrow\!\!\!\rightarrow}$, etc., must be functions of the form $U \longrightarrow \cdots$ and not of the form $U(\mathcal{F}) \longrightarrow \cdots$.

Models and Logical Entailment

The meaning of the formulae $\varphi \vee \psi$, $\varphi \wedge \psi$, and $\neg\varphi$ is defined in the standard way: $\mathbf{I} \models_\nu \varphi \vee \psi$ (or $\mathbf{I} \models_\nu \varphi \wedge \psi$, or $\mathbf{I} \models_\nu \neg\varphi$) if and only if $\mathbf{I} \models_\nu \varphi$ or $\mathbf{I} \models_\nu \psi$ (resp., $\mathbf{I} \models_\nu \varphi$ and $\mathbf{I} \models_\nu \psi$; resp., $\mathbf{I} \not\models_\nu \varphi$). The meaning of the quantifiers is also standard: $\mathbf{I} \models_\nu \psi$, where $\psi = (\forall X)\varphi$ (or $\psi = (\exists X)\varphi$), if and only if $\mathbf{I} \models_\mu \varphi$ for every (resp., some) μ that agrees with ν everywhere, except, possibly, on X.

For a closed formula, ψ, we can omit the mention of ν and simply write $\mathbf{I} \models \psi$, since the meaning of a closed formula is independent of the choice of variable assignments.

An F-structure, **I**, is a *model* of a closed formula, ψ, if and only if $\mathbf{I} \models \psi$. If **S** is a set of formulae and φ is a formula, we write $\mathbf{S} \models \varphi$ (read: φ is logically *implied* or *entailed* by **S**) if and only if φ is true in every model of **S**.

6 Predicates and their Semantics

It is sometimes convenient to have usual first-order predicates on a par with objects (cf. [35, 58, 60, 3]), as certain things are easier to specify in a value-based setting and because experience shows that using predicates alongside with objects leads to more natural specifications. For instance, a program may manipulate mostly objects but, occasionally, it may need to check if a symmetric relationship (*e.g.*, equality or proximity) holds among these objects. Although relationships can always be encoded via objects, these encodings may be contrived or cumbersome. In this subsection, we first explain how predicates can be encoded as F-molecules. Then we show that, with only minor extensions to F-logic, we can incorporate predicates directly, both into the syntax and the semantics.

Predicates can be simulated in several different ways. The approach described here is an adaptation from [58, 60]. To encode an n-ary predicate symbol, p, we introduce a new class for which we will conveniently reuse the same symbol, p. Let *p-tuple* be a new n-ary function symbol. We then assert $(\forall X_1 \ldots \forall X_n)(\textit{p-tuple}(X_1, \ldots, X_n) : p)$ and write classical atoms of the form $p(T_1, \ldots, T_n)$ as molecules of the form:

$$\textit{p-tuple}(T_1, \ldots, T_n)[arg_1 \to T_1; \ldots ; arg_n \to T_n] \qquad (2)$$

It is easily seen that the oid template, $\textit{p-tuple}(T_1, \ldots, T_n)$, is *value-based* in the sense of [102], that is, it is uniquely determined by the values of its attributes. The term $\textit{p-tuple}(T_1, \ldots, T_n)$ can be viewed as an oid of the object that represents a p-relationship among $T_1, \ldots, T_n$, if such a relationship exists; Statement (2) above actually *asserts* that there is a p-relationship among $T_1, \ldots, T_n$.

To incorporate predicates directly, we can extend the notion of F-language with a new set, $\wp$, of *predicate symbols.* If $p \in \wp$ is an n-ary predicate symbol and $T_1, \ldots, T_n$ are id-terms, then $p(T_1, \ldots, T_n)$ is a *predicate molecule* (abbr., P-molecule).

To avoid confusion between id-terms and P-molecules, we assume that predicates and function symbols belong to two disjoint sets of symbols, as in classical logic. In a practical logic programming language, however, this restriction may not be necessary and, as in Prolog, there are advantages in overloading symbols by letting them be used as predicates and as function symbols at the same time (see Section 17.3).

A (generalized) *molecular formula* is now either an F-molecule in the old sense, or a P-molecule (including the case of the equality predicate, *e.g.*, $T \doteq S$).[10] A *literal* is either a molecular formula or a negated molecular formula.[11]

Predicate symbols are interpreted as relations on U using the function $I_\wp$, which, from now on, becomes part of the definition of F-structures:

- $I_\wp(p) \subseteq U^n$, for any n-ary predicate symbol $p \in \wp$.

Given an F-structure $\mathbf{I} = \langle U, \prec_U, \in_U, I_\mathcal{F}, I_\wp, I_\to, I_{\to\!\!\to}, I_{\bullet\!\to}, I_{\bullet\!\to\!\!\to}, I_\Rightarrow, I_{\Rightarrow\!\!\Rightarrow} \rangle$ and a variable assignment ν, we write $\mathbf{I} \models_\nu p(T_1, \ldots, T_n)$ if

- $\langle \nu(T_1), \ldots, \nu(T_n) \rangle \in I_\wp(p)$.

We also fix a diagonal interpretation for the equality predicate:

- $I_\wp(\doteq) \overset{\text{def}}{=} \{\langle a, a \rangle | a \in U\}$.

This, obviously, implies that if T and S are id-terms, then $\mathbf{I} \models_\nu T \doteq S$ if and only if $\nu(T) = \nu(S)$.

[10] So, the term "F-molecule" will now include P-molecules.
[11] In this paper, we do not deal with signatures of P-molecules, as they are studied in [59, 108].

To fully integrate P-molecules into F-logic, we now let F-formulae to be built using all kinds of molecules introduced so far. Truth of formulas in F-structures and the notion of a model is defined as before.

Predicates serve another useful purpose: classical predicate calculus can now be viewed as a subset of F-logic. This has a very practical implication: classical logic programming, too, can be thought of as a special case of programming in F-logic. With some additional effort, most of classical logic programming theory can be adapted to F-logic, which would make it upward-compatible with the existing systems. Section 12 and Appendix A make a first step in this direction.

Another, complimentary way of incorporating predicates into F-logic is to attach them to data molecules syntactically, like methods. As with our earlier integration scheme, these *predicative methods* do not increase the expressive power of the language, but they may lead to more natural representation. For instance,

$$Point[position \rightarrow P;\ inside(Shape)]$$

can be viewed as a shorthand for $Point[position \rightarrow P] \wedge inside(Point, Shape)$ or for $Point[position \rightarrow P;\ inside@Shape \rightarrow true]$. The latter representation also suggests that much of the theory developed for methods (in the following sections), including behavioral inheritance (Section 15.2.2), is applicable to predicative methods.

7 Properties of F-structures

To get a better grip on the notions developed in the previous sections, we present a number of simple, yet important, lemmas about F-structures that directly follow from Definition 5.1 and from other definitions in Sections 5 and 6. We express these properties as assertions about logical entailment, "$\models$", assertions that are true in an arbitrary F-structure, **I**. For simplicity, each assertion deals with ground formulas only and is an immediate consequence of the definitions; easy proofs are left as an exercise. In Section 11, we shall see that the properties presented herein form the basis for F-logic's inference system.

7.1 Properties of the Equality

The lemmas, below, express the usual properties of equality. Together, they express the fact that "$\doteq$" is a congruence relation on $U(\mathcal{F})$.

Reflexivity:

- For all $p \in U(\mathcal{F})$, $\mathbf{I} \models p \doteq p$

Symmetry:

- If $\mathbf{I} \models p \doteq q$ then $\mathbf{I} \models q \doteq p$

Transitivity:

- If $\mathbf{I} \models p \doteq q$ and $\mathbf{I} \models q \doteq r$ then $\mathbf{I} \models p \doteq r$

Substitution:

- If $\mathbf{I} \models s \doteq t \wedge L$, and L' is obtained by replacing an occurrence of s in L with t, then $\mathbf{I} \models L'$

7.2 Properties of the IS-A Relationship

The following states that "::" specifies a partial order on $U(\mathcal{F})$.

IS-A reflexivity:
- $\mathbf{I} \models p :: p$

IS-A transitivity:
- If $\mathbf{I} \models p :: q$ and $\mathbf{I} \models q :: r$ then $\mathbf{I} \models p :: r$

IS-A acyclicity:
- If $\mathbf{I} \models p :: q$ and $\mathbf{I} \models q :: p$ then $\mathbf{I} \models p \doteq q$

Subclass inclusion:
- If $\mathbf{I} \models p : q$ and $\mathbf{I} \models q :: r$ then $\mathbf{I} \models p : r$

The first three claims are direct consequences of the fact that the relation $\prec_U$ on the domain of $\mathbf{I}$ is a partial order; the last property follows from the interplay between $\prec_U$ and $\in_U$.

7.3 Properties of Signature Expressions

In the following properties, the symbol $\approx\!>$ denotes either $\Rightarrow$ or $\Rightarrow\!\!\!>$. Also, in the first two rules, the symbol s stands for an element of $U(\mathcal{F})$ or for "( )" — the empty conjunction of types.

Type inheritance:
- If $\mathbf{I} \models p[mthd\,@\,q_1, \ldots, q_k \approx\!> s]$ and $\mathbf{I} \models r :: p$ then $\mathbf{I} \models r[mthd\,@\,q_1, \ldots, q_k \approx\!> s]$

Input restriction:
- If $\mathbf{I} \models p[mthd\,@\,q_1, \ldots, q_i, \ldots, q_k \approx\!> s]$ and $\mathbf{I} \models q_i' :: q_i$ then $\mathbf{I} \models p[mthd\,@\,q_1, \ldots, q_i', \ldots, q_k \approx\!> s]$

Output relaxation:
- If $\mathbf{I} \models p[mthd\,@\,q_1, \ldots, q_k \approx\!> r]$ and $\mathbf{I} \models r :: s$ then $\mathbf{I} \models p[mthd\,@\,q_1, \ldots, q_k \approx\!> s]$

The first two claims follow directly from the anti-monotonicity constraint on the mappings $I_{\Rightarrow}(\nu(mthd))$ and $I_{\Rightarrow\!\!\!>}(\nu(mthd))$ in Section 5.1.2, where ν is a variable assignment. The last property follows from the upward-closure of $I_{\Rightarrow}(\nu(mthd))(\nu(p), \nu(q_1), \ldots, \nu(q_k))$ and $I_{\Rightarrow\!\!\!>}(\nu(mthd))(\nu(p), \nu(q_1), \ldots, \nu(q_k))$.

Structural Inheritance and Typing

The above properties of signatures are quite interesting and merit further comments. The first of these—*type inheritance* (or *structural inheritance*)—states that structure propagates down from classes to subclasses. In F-logic, this inheritance is *monotonic* in the sense that any additional structure specified for a subclass is *added* to the structure inherited from a superclass. Moreover, structure inherited from different superclasses "adds up." For instance, suppose a class, *cl*, has several signature expressions for the same method, *mthd*. These signatures can be specified directly or inherited by *cl*. Then, "adding up" means that—as a function invoked on a member of class *cl* — the method *mthd* belongs to the intersection of all functional types determined by all signatures (whether specified directly or inherited) that m has in cl.

To illustrate, consider the following example:

$$\begin{array}{ll} empl :: person & person\,[name \Rightarrow string] \\ assistant :: student & student\,[drinks \Rightarrow\!\!\!> beer; drives \Rightarrow bargain] \\ assistant :: empl & empl\,[salary \Rightarrow int; drives \Rightarrow car] \\ & assistant\,[drives \Rightarrow oldThing] \end{array}$$

The signature accumulated by *assistant* from all sources will then be as follows:

$$assistant\,[name \Rightarrow string;\ drinks \Rightarrow\!\!\!> beer;\ drives \Rightarrow\ (bargain, car, oldThing);\ salary \Rightarrow integer]$$

In other words, an *assistant* inherits the type of *name* from the class *person*, the type of *salary* from *employee,* and his drinking habits come from the class *student.* The structure of the attribute *drives* is determined by three factors: the explicitly given signature, $drives \Rightarrow oldThing$; and the two signatures inherited from classes *student* and *empl.* The resulting signature states that assistants drive old cars bought at bargain prices.

We should note that the additive, monotonic signature inheritance of the kind used in F-logic is not applicable in all cases. In fact, most object-oriented systems allow overriding of signatures under certain circumstances. For instance, in the above example, suppose we had $richStudent : student$ and $richStudent\,[drives \Rightarrow expensiveCar]$. Under the monotonic inheritance of F-logic, we would derive $richStudent\,[drives \Rightarrow (expensiveCar, bargain)]$. Since an expensive car of the kind a rich student might drive is not always a bargain, the inheritance of $drives \Rightarrow bargain$ from *student* to *richStudent* leads to unintended results. A more appropriate action in such a case would be to block the inheritance of $drives \Rightarrow bargain$.

Nevertheless, monotonic signature inheritance is appropriate in a wide variety of cases and it is much easier to formalize than signature overriding. It is easy to see that monotonic inheritance can be blocked by giving up the anti-monotonicity property of the mappings $I_{\Rightarrow}(u)$ and $I_{\Rightarrow\!\!\!>}(u)$ on the *first argument* (see Section 5.1.2). Non-monotonic signature inheritance can then be achieved through a technique analogous to (but simpler than) the one developed in Section 15.

The next two properties of signature expressions, *input restriction* and *output relaxation*, say that when methods are viewed as functions, they have the usual properties as far as typing is concerned. For instance, input restriction says that if a method, *mthd*, returns values of class s when it is passed arguments of certain types, then *mthd* will still return values of class s when invoked with arguments of more restricted types. Similarly, output relaxation states that if *mthd* returns values from some class, r, then these values also belong to every superclass s of r. Output relaxation may seem like an obvious and redundant property but, strictly speaking, it does not logically follow from other properties.

7.4 Miscellaneous Properties

The first lemma in this category simply states that scalar methods are supposed to return at most one value for any given set of arguments. The second statement says that object molecules that assert no properties are trivially true in all F-structures. The last claim is the *raison d'être* for the name "molecular formula." According to this property, a molecule that is true in **I** may spin off a bunch of other, simpler formulas that are also true in **I**. Moreover, these formulas cannot be decomposed further and, in a sense, they can be viewed as subformulas of the original molecule.

Scalarity:
- If $\mathbf{I} \models p[scalM \,@\, q_1, \ldots, q_k \rightarrow r_1]$ and $\mathbf{I} \models p[scalM \,@\, q_1, \ldots, q_k \rightarrow r_2]$, then $\mathbf{I} \models r_1 \doteq r_2$
- If $\mathbf{I} \models p[scalM \,@\, q_1, \ldots, q_k \bullet\!\!\rightarrow r_1]$ and $\mathbf{I} \models p[scalM \,@\, q_1, \ldots, q_k \bullet\!\!\rightarrow r_2]$, then $\mathbf{I} \models r_1 \doteq r_2$

Trivial object-molecules:
- For every id-term, t, we have $\mathbf{I} \models t[\,]$

Constituent atoms:
- For every molecule, u, $\mathbf{I} \models u$ if and only if $\mathbf{I} \models v$ holds for every *constituent atom* v of u

The *constituent atoms* mentioned in the last property are, essentially, the indivisible submolecules of u; they are formally defined as follows (for later use, we define constituent atoms for arbitrary molecules, not just the ground ones):

- Every P-molecule or an is-a assertion is its own constituent atom;
- For an object-molecule, $G = P[$ *a ';'-separated list of method expressions* $]$, the constituent atoms are:
  - For every signature expression $Mthd \,@\, Q_1, \ldots, Q_k \approx\!> (R_1, \ldots, R_n)$ in G, where $\approx\!>$ is either $\Rightarrow$ or $\Rightarrow\!\!\!\Rightarrow$, the corresponding constituent atoms are:

$$P[Mthd \,@\, Q_1, \ldots, Q_k \approx\!> (\,)]; \quad \text{and}$$
$$P[Mthd \,@\, Q_1, \ldots, Q_k \approx\!> R_i], \quad i = 1, \ldots, n$$

  - For every scalar data expression $Mthd \,@\, Q_1, \ldots, Q_k \leadsto S$ in G, where $\leadsto$ is either $\rightarrow$ or $\bullet\!\!\rightarrow$, the corresponding constituent atom is:

$$P[Mthd \,@\, Q_1, \ldots, Q_k \leadsto S]$$

  - For every set-valued data expression, $Mthd \,@\, Q_1, \ldots, Q_k \leadsto \{T_1, \ldots, T_m\}$ in G, where $\leadsto$ is either $\twoheadrightarrow$ or $\bullet\!\!\twoheadrightarrow$, the constituent atoms are:

$$P[Mthd \,@\, Q_1, \ldots, Q_k \leadsto \{\,\}] \quad \text{and}$$
$$P[Mthd \,@\, Q_1, \ldots, Q_k \leadsto T_j], \quad j = 1, \ldots, m$$

The property of scalarity follows from the simple fact that scalar invocations of methods are interpreted via single-valued functions. The other claims are straightforward from Definition 5.1.

8 Skolemization and Clausal Form

As in classical logic, the first step in developing a resolution-based proof theory is to convert all formulas into the prenex normal form and then to *Skolemize* them. Skolemized formulas are then transformed into an equivalent *clausal form.*

It is easy to verify that the usual De Morgan's laws hold for F-formulas. Therefore, every F-logic formula has a prenex normal form.

Once a formula is converted into an equivalent prenex normal form, it is Skolemized. Skolemization in F-logic is analogous to the classical case, since id-terms are identical to terms in predicate calculus and because quantification is defined similarly in both cases. For instance, a Skolem normal form for the molecule $(\forall X \exists Y) g(X,Y)[Y \rightarrow f(X,Y);\ a \twoheadrightarrow \{X,Y\}]$ would be $(\forall X) g(X,h(X))[h(X) \rightarrow f(X,h(X));\ a \twoheadrightarrow \{X, h(X)\}]$, where h is a new unary function symbol.

Theorem 8.1 (cf. Skolem's Theorem) *Let φ be an F-formula and φ' be its Skolemization. Then φ is unsatisfiable if and only if so is φ'.*

The proof of this theorem is virtually identical to the standard proof for predicate calculus, and so it is omitted. From now on, we assume that all formulas are Skolemized. De Morgan's Laws further ensure that every formula has a conjunctive and a disjunctive normal form. We can therefore transform every Skolemized formula into a logically equivalent set of clauses, where a *clause* is a disjunction of literals.

9 Herbrand Structures

Given an F-language, $\mathcal{L}$, with $\mathcal{F}$ as its sets of function symbols and $\wp$ as the set of predicate symbols, the *Herbrand universe* of $\mathcal{L}$ is $U(\mathcal{F})$, the set of all ground id-terms. The *Herbrand base* of $\mathcal{L}$, $\mathcal{HB}(\mathcal{L})$, is the set of all ground molecules (including P-molecules and equality).

Let $\mathbf{H}$ be a subset of $\mathcal{HB}(\mathcal{L})$; it is a *Herbrand structure* (abbr., *H-structure*) of $\mathcal{L}$ if it is closed under the logical implication, "$\models$", introduced in Section 5.2. The requirement of $\models$-closure is convenient because ground molecules may imply other molecules in a non-trivial way. For instance, $\{a :: b, b :: c\} \models a :: c$ and $\{a :: b, d[m \Rightarrow a]\} \models d[m \Rightarrow b]$. This is reminiscent of the situation in predicate calculus with equality, where sets of ground atomic formulas may imply other atomic formulas (*e.g.*, $\{a \doteq b,\ p(a)\} \models p(b)$) and where closure with respect to congruence is used for similar reasons.

Since H-structures are $\models$-closed, it is easy to see that they have *closure properties* similar to those in Section 7. These closure properties are obtained from the properties in Section 7 by replacing each statement of the form $\mathbf{I} \models \phi$ with a statement of the form $\phi \in \mathbf{H}$, where ϕ is a molecule, $\mathbf{I}$ is an F-structure, and $\mathbf{H}$ is an Herbrand structure. For instance, the transitivity property for "$\doteq$",

$$\text{If } \mathbf{I} \models p \doteq q \text{ and } \mathbf{I} \models q \doteq r, \text{ then } \mathbf{I} \models p \doteq r$$

now becomes

$$\text{If } p \doteq q \in \mathbf{H} \text{ and } q \doteq r \in \mathbf{H}, \text{ then } p \doteq r \in \mathbf{H}$$

We can now define truth and logical entailment in H-structures, which we shall do only for sets of clauses.

Definition 9.1 (Satisfaction of Formulas by H-structures) Let $\mathbf{H}$ be an H-structure. Then:

- A ground molecule, t, is *true* in $\mathbf{H}$, denoted $\mathbf{H} \models t$, if and only if $t \in \mathbf{H}$;
- A ground negative literal, $\neg t$, is *true* in $\mathbf{H}$, *i.e.*, $\mathbf{H} \models \neg t$, if and only if $t \notin \mathbf{H}$;
- A ground clause, $L_1 \vee \cdots \vee L_n$, is *true* in $\mathbf{H}$ if and only if at least one L_i is true in $\mathbf{H}$;
- A clause, C, is *true* in $\mathbf{H}$ if and only if all ground instances of C are true in $\mathbf{H}$.

If every clause in a set of clauses, $\mathbf{S}$, is true in $\mathbf{H}$, we say that $\mathbf{H}$ is a *Herbrand model* (or an *H-model*) of $\mathbf{S}$. $\square$

Correspondence between H-structures and F-structures

The definition of H-structures indicates that, as in classical logic, there might be a simple way to construct F-structures out of H-structures, and vice versa. One minor problem here is that the Herbrand universe—the domain of all H-structures—cannot always serve as a domain of an F-structure. Indeed, in F-structures, different domain elements represent different objects. However, this is not the case with Herbrand universes. For instance, if $john \doteq father(mary)$ belongs to an H-structure, then the terms $john$ and $father(mary)$ represent the *same* object, yet they are different elements of the Herbrand universe.

The same phenomenon is encountered in classical logic with equality, and the cure for this problem is well-known. Indeed, from Section 7 and the remarks above, it follows that equality is a congruence relation on the Herbrand universe. So, we can construct a domain of an F-structure by factoring $U(\mathcal{F})$ with this relation.

The correspondence between H-structures and F-structures can now be stated as follows: Given an F-structure for a set of clauses **S**, the corresponding H-structure is the set of ground molecules that are true in the F-structure. Conversely, for an H-structure, **H**, the corresponding F-structure, $\mathbf{I}_H = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\wp}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\rightarrow}, I_{\bullet\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rrightarrow} \rangle$, is defined as follows:[12]

1. The domain, U, is $U(\mathcal{F})/\doteq$ — the quotient of $U(\mathcal{F})$ induced by the equalities in **H**. We denote the equivalence class of t by $[t]$.
2. The ordering, $\prec_U$, and the class membership relation, $\in_U$, are determined by the is-a assertions in **H**: For all $[t], [s] \in U$, we assert $[s] \preceq_U [t]$ if and only if $s :: t \in \mathbf{H}$; and $[s] \in_U [t]$ if and only if $s : t \in \mathbf{H}$.
3. $I_{\mathcal{F}}(c) = [c]$, for every 0-ary function symbol $c \in \mathcal{F}$.
4. $I_{\mathcal{F}}(f)([t_1], \ldots, [t_k]) = [f(t_1, \ldots, t_k)]$, for every k-ary ($k \geq 1$) function symbol $f \in \mathcal{F}$.
5. $I^{(k)}_{\rightarrow}([scalM])([obj], [t_1], \ldots, [t_k]) = \begin{cases} [s] & \text{if } obj\,[scalM \,@\, t_1, \ldots, t_k \rightarrow s] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$
6. $I^{(k)}_{\twoheadrightarrow}([setM])([obj], [t_1], \ldots, [t_k]) =$

$$\begin{cases} \{[s] \mid \text{where } obj\,[setM \,@\, t_1, \ldots, t_k \twoheadrightarrow s] \in \mathbf{H}\} & \text{if } obj\,[setM \,@\, t_1, \ldots, t_k \twoheadrightarrow \{\,\}] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The mappings $I_{\bullet\!\rightarrow}$ and $I_{\bullet\!\twoheadrightarrow}$ are defined similarly to (5) and (6), except that inheritable data expressions must be used instead of the non-inheritable ones.

7. $I^{(k)}_{\Rightarrow}([scalM])([obj], [t_1], \ldots, [t_k]) =$

$$\begin{cases} \{[s] \mid \text{where } obj\,[scalM \,@\, t_1, \ldots, t_k \Rightarrow s] \in \mathbf{H}\} & \text{if } obj\,[scalM \,@\, t_1, \ldots, t_k \Rightarrow (\,)] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

8. $I^{(k)}_{\Rrightarrow}([setM])([obj], [t_1], \ldots, [t_k]) =$

$$\begin{cases} \{[s] \mid \text{where } obj\,[setM \,@\, t_1, \ldots, t_k \Rrightarrow s] \in \mathbf{H}\} & \text{if } obj\,[setM \,@\, t_1, \ldots, t_k \Rrightarrow (\,)] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

[12] Observe the similarity with the corresponding construction in classical logic with equality.

9. $I_{\wp}(p) \;=\; \{\langle [t_1], \ldots, [t_k] \rangle \mid p(t_1, \ldots, t_k) \in \mathbf{H}\}$.

We remark that, in (6), there is a difference between the set $I_{\rightarrow}^{(k)}([setM])([obj], [t_1], \ldots, [t_k])$ being undefined and being empty. It is undefined if $\mathbf{H}$ contains no atoms of the form $obj\,[setM\,@\,t_1, \ldots, t_k \twoheadrightarrow \ldots]$, not even $obj\,[setM\,@\,t_1, \ldots, t_k \twoheadrightarrow \{\,\}]$. In contrast, $I_{\twoheadrightarrow}^{(k)}([setM])([obj], [t_1], \ldots, [t_k])$ is empty when $\mathbf{H}$ does contain $obj\,[setM\,@\,t_1, \ldots, t_k \twoheadrightarrow \{\,\}]$, but has no atoms of the form $obj\,[setM\,@\,t_1, \ldots, t_k \twoheadrightarrow s]$, for any $s \in U(\mathcal{F})$. Similar remarks apply to $I_{\Rightarrow\mkern-14mu\Rightarrow}^{(k)}$ in (7) and to $I_{\Rightarrow\mkern-14mu\Rightarrow}^{(k)}$ in (8).

It is easy to see that $\mathbf{I}_H = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\wp}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\rightarrow}, I_{\bullet\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\mkern-14mu\Rightarrow} \rangle$ is well-defined and is, indeed, an F-structure. The above correspondence immediately leads to the following result:

Proposition 9.2 *Let* $\mathbf{S}$ *be a set of clauses. Then* $\mathbf{S}$ *is unsatisfiable if and only if* $\mathbf{S}$ *has no H-model.*

Proof: It is easy to verify that for every H-structure, $\mathbf{H}$, the entailment $\mathbf{H} \models \mathbf{S}$ takes place if and only if $\mathbf{I}_H \models \mathbf{S}$, where $\mathbf{I}_H$ is the F-structure that corresponds to $\mathbf{H}$, as defined earlier. □

10 Herbrand's Theorem

In classical logic, a set of clauses, $\mathbf{S}$, is unsatisfiable if and only if so is some finite set of ground instances of clauses in $\mathbf{S}$; this property is commonly referred to as *Herbrand's theorem.* In F-logic, Herbrand's theorem plays the same fundamental role. We establish this theorem by considering *maximal finitely satisfiable* sets, similarly to the proof of the compactness theorem in [40]. A set $\mathbf{S}$ of ground clauses is *finitely satisfiable*, if every finite subset of $\mathbf{S}$ is satisfiable. A finitely satisfiable set $\mathbf{S}$ is *maximal*, if no other set of ground clauses containing $\mathbf{S}$ is finitely satisfiable. Some useful properties of finitely satisfiable sets are stated below.

Lemma 10.1 *Given a finitely satisfiable set of ground clauses,* $\mathbf{S}$*, there exists a maximal finitely satisfiable set,* $\mathbf{T}$*, such that* $\mathbf{S} \subseteq \mathbf{T}$.

Proof: Let Λ be a collection of all finitely satisfiable sets of ground clauses (in a fixed language $\mathcal{L}$) that contain $\mathbf{S}$. The set Λ is partially ordered by set-inclusion. Since $\mathbf{S} \in \Lambda$, Λ is non-empty. Furthermore, for every $\subseteq$-growing chain $\Sigma \subseteq \Lambda$, the least upper bound of the chain, $\cup\Sigma$, is also in Λ. Indeed:

- $\cup\Sigma$ contains $\mathbf{S}$; and
- $\cup\Sigma$ is finitely satisfiable (for, if not, one of the elements of Σ must not be finitely satisfiable).

By Zorn's Lemma, Λ has a maximal element. □

Lemma 10.2 *Let* $\mathbf{T}$ *be a maximal finitely satisfiable set of ground clauses.*

(i) For every ground F-molecule T*, either* $T \in \mathbf{T}$ *or* $\neg T \in \mathbf{T}$.

(ii) A ground clause, $L_1 \vee \cdots \vee L_n$*, is in* $\mathbf{T}$ *if and only if* $L_i \in \mathbf{T}$*, for some* $1 \leq i \leq n$.

Proof: Part (i): If $\mathbf{T}$ is finitely satisfiable, then so is $\mathbf{T} \cup \{T\}$ or $\mathbf{T} \cup \{\neg T\}$. Therefore, $\mathbf{T}$ must contain either T or $\neg T$, since it is maximal. Part (ii) is proved similarly to (i). □

Lemma 10.3 *Let* $\mathbf{T}$ *be a maximal finitely satisfiable set of ground clauses. Let* $\mathbf{H}$ *be the set of all ground molecules in* $\mathbf{T}$. *Then* $\mathbf{H}$ *is an H-structure.*

Proof: The set $\mathbf{H}$ is $\models$-closed, since so is $\mathbf{T}$ (or else $\mathbf{T}$ is not maximal). □

Theorem 10.4 (cf. Herbrand's Theorem)
A set of clauses, $\mathbf{S}$*, is unsatisfiable if and only if so is some finite subset of ground instances of the clauses in* $\mathbf{S}$.

Proof: For the "if" part, assume that some finite subset of ground clauses of $\mathbf{S}$ is unsatisfiable. Then $\mathbf{S}$ is also unsatisfiable. The "only-if" part is proved by contradiction. Let $\mathbf{S}'$ denote the set of all ground instances of the clauses in $\mathbf{S}$, and suppose that all finite subsets of $\mathbf{S}'$ are satisfiable, which implies that $\mathbf{S}'$ itself is finitely satisfiable. We will show that then $\mathbf{S}$ is satisfiable.

By Lemma 10.1, $\mathbf{S}'$ can be extended to a maximal finitely satisfiable set, $\mathbf{T}$. Let $\mathbf{H}$ be the set of all ground molecules in $\mathbf{T}$, which is an H-structure, by Lemma 10.3. We claim that $\mathbf{H} \models C$ if and only if $C \in \mathbf{T}$, for every ground clause, C. Consider the following cases:

(a) *C is a ground molecule.* By definition, $\mathbf{H} \models C$ if and only if $C \in \mathbf{T}$.

(b) *C is a negative literal $\neg P$.* Then $\mathbf{H} \models \neg P$ if and only if $P \notin \mathbf{H}$. Since $\mathbf{H}$ contains all the ground molecules in $\mathbf{T}$, $P \notin \mathbf{H}$ if and only if $P \notin \mathbf{T}$. Finally, by (i) of Lemma 10.2, $P \notin \mathbf{T}$ if and only if $\neg P \in \mathbf{T}$.

(c) *C is a disjunction of ground literals $L_1 \vee \cdots \vee L_n$.* Then
$\mathbf{H} \models L_1 \vee \cdots \vee L_n$
if and only if $\mathbf{H} \models L_i$ for some i, by definition;
if and only if $L_i \in \mathbf{T}$, by cases (a) and (b) above;
if and only if $L_1 \vee \cdots \vee L_n \in \mathbf{T}$, by (ii) of Lemma 10.2.

We have shown that $\mathbf{H}$ satisfies every clause of $\mathbf{T}$. Since $\mathbf{S}' \subseteq \mathbf{T}$, $\mathbf{H}$ is an H-model of $\mathbf{S}$. By Proposition 9.2, $\mathbf{S}$ is satisfiable. □

Herbrand's Theorem is a basis for the resolution-based proof theory in classical logic [31]. In the next section we use Herbrand's Theorem for F-logic to develop a similar result, which extends the results in [60]. Just as classical proof theory was fundamental to the theory of logic programming, we anticipate that the proof theory of F-logic will play a similar role in the object-oriented domain.

11 Proof Theory

This section describes a sound and complete proof theory for the logical entailment relation "$\models$" of Section 5.2. The theory consists of twelve inference rules and one axiom. The rules of resolution, factoring, and paramodulation form the core of the deductive system. However, unlike in predicate calculus, these three rules are not enough. For a deductive system to be complete, additional rules for capturing the properties of types and IS-A hierarchies are needed. The large number of inference rules in F-logic compared to predicate calculus stems from the rich semantics of object-oriented systems; this is likely to be the case with any logical system that attempts to adequately capture this paradigm. As will be seen shortly, many rules are quite similar to each other, except that one may deal with data expressions and the other with signatures or with P-molecules. It is possible to reduce the number of rules by half by increasing the power of the resolution rule. This is analogous to classical logic, where factoring is often combined with resolution. However, we prefer to have many simple rules instead of a few complex ones, because this makes it easier to see the rationale behind each rule.

11.1 Substitutions and Unifiers

In F-logic, much of the theory of unifiers carries over from the classical case. However, this notion needs some adjustments to accommodate sets and other syntactic features of F-molecules.

Substitutions

Let $\mathcal{L}$ be a language with a set of variables $\mathcal{V}$. A *substitution* is a mapping $\sigma : \mathcal{V} \longrightarrow \{\text{id-terms of } \mathcal{L}\}$ such that it is an identity everywhere outside some finite set $dom(\sigma) \subseteq \mathcal{V}$, the *domain* of σ.

As in classical logic, substitutions extend to mappings $\{$ *id-terms* $\} \longrightarrow \{$ *id-terms* $\}$ as follows:

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

A substitution, σ, can be further extended to a mapping from molecules to molecules by distributing σ through molecules' components. For instance, $\sigma(Q : P)$ is defined as $\sigma(Q) : \sigma(P)$, $\sigma(Q :: P)$ as $\sigma(Q) :: \sigma(P)$, and $\sigma(\, Q\,[Mthd\,@\,R, S \rightarrow T]\,)$ is the same as $\sigma(Q)[\sigma(Mthd)\,@\,\sigma(R), \sigma(S) \rightarrow \sigma(T)]$. Similarly, substitutions extend to F-formulae by distributing them through logical connectives and quantifiers.

A substitution is *ground* if $\sigma(X) \in U(\mathcal{F})$ for each $X \in dom(\sigma)$, that is, if $\sigma(X)$ has no variables. Given a substitution σ and a formula φ, $\sigma(\varphi)$ is called an *instance* of φ. It is a *ground* instance if it contains no variables. A formula is *ground* if it has no variables.

Unifiers

Unification of id-terms, is-a molecules, and P-molecules is no different than in classical logic. Let T_1 and T_2 be a pair of id-terms, is-a molecules, or P-molecules. A substitution σ is a *unifier* of T_1 *and* T_2, if $\sigma(T_1) = \sigma(T_2)$. This unifier is *most general*, written $mgu(T_1, T_2)$, if for every unifier μ of T_1 and T_2, there exists a substitution γ, such that $\mu = \gamma \circ \sigma$.

For object molecules, instead of requiring identity under unification we merely ask that unifiers would map molecules into *submolecules* of other molecules.

Definition 11.1 (Asymmetric Unification of Object Molecules) Let $L_1 = S\,[\ldots]$ and $L_2 = S\,[\ldots]$ be a pair of object molecules with the same object id, S.

We say that L_1 is a *submolecule* of L_2, denoted $L_1 \sqsubseteq L_2$, if and only if every constituent atom of L_1 (defined in Section 7) is also a constituent atom of L_2.

A substitution σ is a *unifier* of L_1 *into* L_2 (note the asymmetry!) if and only if $\sigma(L_1) \sqsubseteq \sigma(L_2)$. □

For instance, $L = S[M \,@\, X \rightarrow V]$ is unifiable into $L' = S[N \,@\, Y \rightarrow W; Z \,@\, Y \bullet\!\!\rightarrow T]$ with the unifier $\{M\backslash N, X\backslash Y, V\backslash W\}$, but not the other way around (because the atom $S[Z \,@\, Y \bullet\!\!\rightarrow T]$ cannot be turned into a constituent atom of L). However, a slightly different molecule, $S[N \,@\, Y \rightarrow W; Z \,@\, Y \rightarrow T]$, *is* unifiable into L with the unifier $\{N\backslash M, Y\backslash X, W\backslash V, Z\backslash M, T\backslash V\}$.

Complete Sets of Most General Unifiers

Defining *most general* unifiers for object molecules requires more work. Consider $L_1 = a[set \twoheadrightarrow X]$ and $L_2 = a[set \twoheadrightarrow \{b, c\}]$. Intuitively, there are two unifiers of L_1 into L_2 that can be called "most general:" $X\backslash b$ and $X\backslash c$. Clearly, none of these unifiers is more general than the other and, therefore, the definition of mgu that works for P-molecules and for is-a assertions does not work here. A common approach in such situations is to consider *complete sets* of most general unifiers.

Definition 11.2 (Most General Unifiers) Let L_1 and L_2 be a pair of molecules and let α, β be a pair of unifiers of L_1 into L_2. We say that α is *more general* than β, denoted $\alpha \trianglelefteq \beta$, if and only if there is a substitution γ such that $\beta = \gamma \circ \alpha$. A unifier α of L_1 into L_2 is *most general* (abbr., *mgu*) if for every unifier β, $\beta \trianglelefteq \alpha$ implies $\alpha \trianglelefteq \beta$.

A set Σ of most general unifiers of L_1 into L_2 is *complete* if for every unifier θ of L_1 into L_2 there is $\alpha \in \Sigma$ such that $\alpha \trianglelefteq \theta$. $\square$

Just as there is a unique up to the equivalence mgu in the classical case, it easily follows from the definitions that the complete set of unifiers of L_1 into L_2 is also unique up to the equivalence.[13] An algorithm that computes a complete set of mgu's appears in Appendix C.

In predicate calculus, the notion of a unifier works for an arbitrary number of terms to be unified. Extension of our definitions to accommodate an arbitrary number of id-terms, P-molecules, or is-a assertions is obvious. For object molecules, we say that a substitution σ is a *unifier* of $L_1, \ldots, L_n$ *into* L when $\sigma(L_i) \sqsubseteq L$, for $i = 1, \ldots, n$. Generalization of the notion of mgu is straightforward and is left as an exercise.

For convenience, we also define mgu's for tuples of id-terms. Tuples $\langle P_1, \ldots, P_n\rangle$ and $\langle Q_1, \ldots, Q_n\rangle$ are *unifiable* when there is a substitution σ such that $\sigma(P_i) = \sigma(Q_i)$, $i = 1, \ldots, n$. This unifier is *most general* (written $mgu(\langle P_1, \ldots, P_n\rangle, \langle Q_1, \ldots, Q_n\rangle)$), if for every other unifier, μ, of these tuples, $\mu = \gamma \circ \sigma$ for some substitution γ. It is easy to see that any mgu of $\langle P_1, \ldots, P_n\rangle$ and $\langle Q_1, \ldots, Q_n\rangle$ coincides with the mgu of $f(P_1, \ldots, P_n)$ and $f(Q_1, \ldots, Q_n)$, where f is some n-ary function symbol. Therefore, mgu of a pair of tuples is unique.

In the sequel, $mgu_{\sqsubseteq}(L_1, L_2)$ will be used to denote *some* most general unifier of one molecule, L_1, into another, L_2. As shown earlier, for object molecules $mgu_{\sqsubseteq}(L_1, L_2)$ may exist while $mgu_{\sqsubseteq}(L_2, L_1)$ may not. On the other hand, unification of id-terms, is-a assertions, and P-molecules is a symmetric operation. Nevertheless, we can still talk about unification of one such expression *into* another—a convention that can often simplify the language. Also, to simplify notation, all our inference rules will be based on F-atoms, not F-molecules.

Finally, we remark that, as in predicate calculus, prior to any application of an inference rule, the clauses involved in the application must be *standardized apart*. This means that variables must be

[13]A set of unifiers, Ω_1, is *equivalent* to Ω_2 if for every $\sigma_1 \in \Omega_1$ there is $\sigma_2 \in \Omega_2$ such that $\sigma_2 \trianglelefteq \sigma_1$; and vice versa, for every $\sigma_2 \in \Omega_2$ there is $\sigma_1 \in \Omega_1$ such that $\sigma_1 \trianglelefteq \sigma_2$.

consistently renamed so that the resulting clauses will share none. However, clauses used by an inference rule may be instances of the same clause; they can even be identical, if no variables are involved.

11.2 Core Inference Rules

For simplicity, but without loss of generality, only binary resolution is considered. In the inference rules, below, the symbols L and L' will be used to denote positive literals, C and C' will denote clauses, and P, Q, R, S, T, etc., will denote id-terms.

Resolution: Let $W = \neg L \vee C$ and $W' = L' \vee C'$ be a pair of clauses that are standardized apart. Let θ be an mgu of L into L'. The *resolution* rule is, then, as follows:

$$\textbf{from} \quad W \quad \text{and} \quad W' \quad \textbf{derive} \quad \theta(C \vee C')$$

Notice that when L and L' are object molecules, resolution is *asymmetric* since $\theta = mgu_{\sqsubseteq}(L, L')$ may be different from $mgu_{\sqsubseteq}(L', L)$, and the latter mgu may not even exist. As in the classical case, binary resolution must be complimented with the so-called factoring rule that seeks to reduce the number of disjuncts in a clause.

Factoring: The factoring rule has two forms, depending on the polarity of literals to be factored. For positive literals, consider a clause of the form $W = L \vee L' \vee C$, where L and L' are positive literals. Let L be unifiable *into* L' with the mgu θ. The *factoring* rule is, then, as follows:

$$\textbf{from} \quad W \quad \textbf{derive} \quad \theta(L \vee C)$$

In case of negative literals, if $W = \neg L \vee \neg L' \vee C$ and L is unifiable *into* L' with the mgu θ, then the factoring rule is:

$$\textbf{from} \quad W \quad \textbf{derive} \quad \theta(\neg L' \vee C)$$

Clauses inferred by one of the two factoring rules are called *factors* of W. Note that in both inference rules L must be unifiable into L'. However, in the first case, it is the literal L that survives, while in the second rule it is L'.

To account for the equality relation, we need a paramodulation rule. When there is a need to focus on a specific occurrence of an id-term, T, in an expression, E (which can be a literal or an id-term), it is a standard practice to write E as $E\,[T]$. If one single occurrence of T is replaced by S, the result will be denoted by $E\,[T \backslash S]$.

Paramodulation: Consider a pair of clauses, $W = L[T] \vee C$ and $W' = (T' \doteq T'') \vee C'$, with no common variables. If T and T' are id-terms unifiable with an mgu, θ, then the *paramodulation* rule says:

$$\textbf{from} \quad W \quad \text{and} \quad W' \quad \textbf{derive} \quad \theta(L[T \backslash T''] \vee C \vee C')$$

Resolution: $$\frac{\neg L \vee C, \;\; L' \vee C', \;\; \theta = mgu_{\sqsubseteq}(L, L')}{\theta(C \vee C')}$$

Factoring: $$\frac{L \vee L' \vee C, \;\; \theta = mgu_{\sqsubseteq}(L, L')}{\theta(L \vee C)} \qquad \frac{\neg L \vee \neg L' \vee C, \;\; \theta = mgu_{\sqsubseteq}(L, L')}{\theta(\neg L' \vee C)}$$

Paramodulation: $$\frac{L[T] \vee C, \;\; (T' \doteq T'') \vee C', \;\; \theta = mgu(T, T')}{\theta(L[T \backslash T''] \vee C \vee C')}$$

Figure 5: Summary of the Core Inference Rules

11.3 IS-A Inference Rules

The following axiom and rules capture the semantics of the subclass-relationship and its interaction with class membership.

IS-A reflexivity: The following is the *IS-A reflexivity* axiom:

$$(\forall X) \; X :: X$$

IS-A acyclicity: Let $W = (P :: Q) \vee C$ and $W' = (Q' :: P') \vee C'$ be clauses with no variables in common. Suppose that θ is an mgu of tuples $\langle P, Q \rangle$ and $\langle P', Q' \rangle$ of id-terms. The *IS-A acyclicity* rule is as follows:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta((P \doteq Q) \vee C \vee C')$$

Note that IS-A reflexivity and IS-A acyclicity imply reflexivity of equality. Indeed, since $X :: X$ is an axiom, by IS-A acyclicity, one can derive $X \doteq X$ from $X :: X$ and $X :: X$.

IS-A transitivity: Let $W = (P :: Q) \vee C$ and $W' = (Q' :: R') \vee C'$ be standardized apart and let θ be an mgu of Q and Q'. The *IS-A transitivity* rule, then, is:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta((P :: R') \vee C \vee C')$$

Subclass inclusion: Let $W = (P : Q) \vee C$ and $W' = (Q' :: R') \vee C'$ be standardized apart and let θ be an mgu of Q and Q'. Then the *subclass inclusion* rule says:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta((P : R') \vee C \vee C')$$

11.4 Type Inference Rules

Signature expressions have the properties of type inheritance, input restriction, and output relaxation that are captured by the following inference rules:

IS-A reflexivity:	X::X
IS-A acyclicity:	$\dfrac{(P :: Q) \vee C,\ \ (Q' :: P') \vee C',\ \ \theta = mgu(\langle P, Q\rangle, \langle P', Q'\rangle)}{\theta(\,(P \doteq Q) \vee C \vee C'\,)}$
IS-A transitivity:	$\dfrac{(P :: Q) \vee C,\ \ (Q' :: R') \vee C',\ \ \theta = mgu(Q, Q')}{\theta((P :: R') \vee C \vee C')}$
Subclass inclusion:	$\dfrac{(P : Q) \vee C,\ \ (Q' :: R') \vee C',\ \ \theta = mgu(Q, Q')}{\theta((P : R') \vee C \vee C')}$

Figure 6: Summary of the IS-A Inference Rules

Type inheritance: Let $W = P\,[Mthd\,@\,Q_1, \ldots, Q_k \Rightarrow T] \vee C$ and $W' = (S' :: P') \vee C'$ be a pair of clauses with no common variables, and suppose P and P' have an mgu, θ. The *type inheritance* rule states the following:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(S'[Mthd\,@\,Q_1, \ldots, Q_k \Rightarrow T] \vee C \vee C')$$

In other words, S' inherits the signature of P'. A similar rule exists for set-valued methods. If $W = P\,[Mthd\,@\,Q_1, \ldots, Q_k \Rrightarrow T] \vee C$ and W' is as before, then:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(S'[Mthd\,@\,Q_1, \ldots, Q_k \Rrightarrow T] \vee C \vee C')$$

Input restriction: Let $W = P\,[Mthd\,@\,Q_1, \ldots, Q_i, \ldots, Q_k \Rightarrow T] \vee C$ and $W' = (Q_i'' :: Q_i') \vee C'$ be standardized apart. Suppose also that Q_i and Q_i' have an mgu θ. The *input restriction* rule states:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(P\,[Mthd\,@\,Q_1, \ldots, Q_i'', \ldots, Q_k \Rightarrow T] \vee C \vee C')$$

Here Q_i'' replaces Q_i. A similar rule exists for set-valued methods. Also, it should be noted that T in the above three inference rules stands for "()" or an id-term.

Output relaxation: Consider clauses $W = P\,[Mthd\,@\,Q_1, \ldots, Q_k \Rightarrow R] \vee C$ and $W' = (R' :: R'') \vee C'$ with no common variables, and suppose R and R' have an mgu θ. The *output relaxation* rule, then, states:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(P\,[Mthd\,@\,Q_1, \ldots, Q_k \Rightarrow R''] \vee C \vee C')$$

A similar rule applies to set-valued methods.

11.5 Miscellaneous Inference Rules

The requirement that a scalar method must return at most one value is built into in the following rule:

Scalarity: Consider a pair of clauses that share no common variables:

$$W = P\,[Mthd\,@\,Q_1, \ldots, Q_k \to R] \vee C \quad \text{and} \quad W' = P'[Mthd'\,@\,Q_1', \ldots, Q_k' \to R'] \vee C'.$$

Type inheritance:
$$\frac{P\,[Mthd\,@\,Q_1,\ldots,Q_k \Rightarrow T] \vee C,\ (S' :: P') \vee C',\ \theta = mgu(P,P')}{\theta(S'[Mthd\,@\,Q_1,\ldots,Q_k \Rightarrow T] \vee C \vee C')}$$

Similarly for set-valued methods

Input restriction:
$$\frac{P\,[Mthd\,@\,Q_1,\ldots,Q_i,\ldots,Q_k \Rightarrow T] \vee C,\ (Q''_i :: Q'_i) \vee C',\ \theta = mgu(Q_i,Q'_i)}{\theta(P\,[Mthd\,@\,Q_1,\ldots,Q''_i,\ldots,Q_k \Rightarrow T] \vee C \vee C')}$$

Similarly for set-valued methods

Output relaxation:
$$\frac{P\,[Mthd\,@\,Q_1,\ldots,Q_k \Rightarrow R] \vee C,\ \ (R' :: R'') \vee C',\ \ \theta = mgu(R,R')}{\theta(P\,[Mthd\,@\,Q_1,\ldots,Q_k \Rightarrow R''] \vee C \vee C')}$$

Similarly for set-valued methods

Figure 7: Summary of the Type-Inference Rules

Suppose there is an mgu θ that unifies the tuple of id-terms $\langle P, Mthd, Q_1, \ldots, Q_k\rangle$ with the tuple $\langle P', Mthd', Q'_1, \ldots, Q'_k\rangle$. The rule of *scalarity* then says:

$$\textbf{from}\quad W \quad \text{and} \quad W' \ \ \textbf{derive}\ \ \theta(\,(R \doteq R') \vee C \vee C')$$

A similar rule exists for inheritable scalar expressions. The only difference is that $\rightarrow$ is replaced with $\bullet\!\!\rightarrow$ in W and W'.

Another miscellaneous rule is called *merging*; it seeks to combine information contained in different object molecules. Let L_1 and L_2 be a pair of such molecules with the same object id. An object-molecule L is called a *merge* of L_1 and L_2, if the set of constituent atoms of L is precisely the union of the sets of constituent atoms of L_1 and L_2. A pair of molecules can be merged in several different ways when they have common set-valued methods. For example, the terms

$$T\,[ScalM \rightarrow d;\ SetM \twoheadrightarrow e;\ SetM\,@\,X \twoheadrightarrow b] \tag{3}$$
$$T\,[ScalM \rightarrow g;\ SetM\,@\,Y \twoheadrightarrow h;\ SetM\,@\,X \twoheadrightarrow c] \tag{4}$$

have more than one merge:

$$T\,[ScalM \rightarrow d;\ ScalM \rightarrow g;\ SetM \twoheadrightarrow e;\ SetM\,@\,Y \twoheadrightarrow h;\ SetM\,@\,X \twoheadrightarrow b;\ SetM\,@\,X \twoheadrightarrow c] \tag{5}$$

and

$$T\,[ScalM \rightarrow d;\ ScalM \rightarrow g;\ SetM \twoheadrightarrow e;\ SetM\,@\,Y \twoheadrightarrow h;\ SetM\,@\,X \twoheadrightarrow \{b,c\}] \tag{6}$$

However, we distinguish certain kind of merges that have a uniqueness property. We call them *canonical* merges.

An *invocation* of a method consists of the method's name, its arguments, and the arrow specifying the type of invocation (scalar or set-valued). For instance, in the above example, $ScalM \rightarrow$, $SetM \twoheadrightarrow$,

Scalarity:

$$\frac{\begin{array}{c} P\,[Mthd\,@\,Q_1,\ldots,Q_k \rightarrow R] \vee C_1,\;\; P'[Mthd'\,@\,Q'_1,\ldots,Q'_k \rightarrow R'] \vee C_2 \\ \theta = mgu(\,\langle P, Mthd, Q_1,\ldots,Q_k\rangle,\ \langle P', Mthd', Q'_1,\ldots,Q'_k\rangle\,) \end{array}}{\theta(\,R \doteq R' \vee C_1 \vee C_2)}$$

Similarly for inheritable scalar expressions, where $\rightarrow$ is replaced with $\bullet\!\!\rightarrow$

Merging:

$$\frac{P[\ldots] \vee C,\;\; P'[\ldots] \vee C',\;\; \theta = mgu(P, P'),\;\; L'' = merge(\theta(P[\ldots]), \theta(P'[\ldots]))}{L'' \vee \theta(C \vee C')}$$

Elimination:

$$\frac{\neg P[\,] \vee C}{C}$$

Figure 8: Summary of the Miscellaneous Inference Rules

$SetM\,@\,X \twoheadrightarrow$, and $SetM\,@\,Y \twoheadrightarrow$ are all distinct invocations. A *canonical merge* of L_1 and L_2, denoted $merge(L_1, L_2)$, is a merge that does not contain repeated identical invocations of set-valued methods. In the above, (6) is a canonical merge of (3) and (4). Clearly, $merge(L_1, L_2)$ is unique up to a permutation of atoms and id-terms in the ranges of set-valued methods.

Merging: Consider a pair of standardized apart clauses, $W = L \vee C$ and $W' = L' \vee C'$, where both L and L' are object molecules. Let θ be an mgu unifying the oid parts of L and L'. Let L'' denote the canonical merge of $\theta(L)$ and $\theta(L')$. The *merging* rule, then, sanctions the following derivation:

$$\textbf{from } W \text{ and } W' \textbf{ derive } L'' \vee \theta(C \vee C')$$

Finally, since for every id-term, P, the molecule $P\,[\,]$ is a tautology, we have the following *elimination* rule:

Elimination: If C is a clause and P an id-term then:

$$\textbf{from } \neg P[\,] \vee C \textbf{ derive } C$$

Notice that if C is an empty clause then the elimination rule would derive an empty clause as well.

11.6 Remarks

With such multitude of inference rules, a natural concern is whether there can be an efficient evaluation procedure for F-logic queries. The answer to this question is believed to be positive: several major evaluation strategies developed for deductive databases (*e.g.*, OLDT [98, 38, 106] or Magic Sets [19, 91]) are applicable here as well.

Another important point is that one does not need to use some of the inference rules at run time. For instance, in proof-theoretic terms, the purpose of static type checking is to obviate the need for the typing rules at run time. Likewise, a compile-time algorithm for checking acyclicity could be used to get rid of the IS-A acyclicity rule at run time. A practical system is also likely to limit the use of the rule of scalarity. For instance, this rule may be used to generate run-time warnings regarding possible inconsistencies detected in scalar methods, but it may not be used to do inference.

11.7 Soundness of the Proof Theory

Given a set, **S**, of clauses, a *deduction* of a clause, C, from **S** is a finite sequence of clauses $D_1, \ldots, D_n$ such that $D_n = C$ where, for $1 \leq k \leq n$, D_k is either

- a member of **S**, or
- is derived from some D_i and, possibly, an additional clause, D_j, where $i, j < k$, using one of the core, is-a, type, or miscellaneous inference rules.

A deduction ending with the empty clause, $\Box$, is called a *refutation* of **S**. If C is deducible from **S**, we shall write $\mathbf{S} \vdash C$.

Theorem 11.3 (Soundness of F-logic Deduction) *If* $\mathbf{S} \vdash C$ *then* $\mathbf{S} \models C$.

Proof: Directly follows from the closure properties given in Section 7 and from the form of the inference rules. $\Box$

11.8 A Sample Proof

Consider the following set of clauses:

i.	$a :: b$	iv.	$r\,[attr \rightarrow a]$
ii.	$p(a)$	v.	$r\,[attr \rightarrow f(S)] \vee \neg p(X) \vee \neg O\,[M\,@\,X \Rightarrow S]$
iii.	$c\,[m\,@\,b \Rightarrow (v, w)]$	vi.	$\neg p(f(Z))$

We can refute the above set using the following sequence of derivation steps, where θ denotes the unifier used in the corresponding step:

vii.	$r\,[attr \rightarrow f(S)] \vee \neg O\,[M\,@\,a \Rightarrow S]$	by resolving (ii) and (v); $\theta = \{X \backslash a\}$
viii.	$c\,[m\,@\,a \Rightarrow (v, w)]$	by input restriction from (i) and (iii)
ix.	$r\,[attr \rightarrow f(v)]$	by resolving (viii) with (vii); $\theta = \{O\backslash c, S\backslash v, M\backslash m\}$
x.	$a \doteq f(v)$	by the rule of scalarity, using (iv) and (ix)
xi.	$p(f(v))$	by paramodulation, using (ii) and (x)
xii.	$\Box$	by resolving (vi) with (xi); $\theta = \{Z \backslash v\}$

All steps in this derivation are self-explanatory. We would like to point out, though, that θ in Step (ix) is an asymmetric unifier that unifies the atom $O\,[M\,@\,a \Rightarrow S]$ *into* the molecule (viii).

11.9 Completeness of the Proof Theory

We follow the standard strategy for proving completeness, adapted from classical logic. First, Herbrand's Theorem is used to establish completeness for the ground case. Then, an analogue of Lifting Lemma shows that ground refutations can be "lifted" to the nonground case.

Lemma 11.4 *Let* **S** *be a set of ground F-literals. If* **S** *is unsatisfiable then there are molecules P and Q such that $P \sqsubseteq Q$, $\neg P \in$* **S** *and* $\mathbf{S} \vdash Q$. *(When P, Q are P-molecules or is-a assertions, $P \sqsubseteq Q$ should be taken to mean that P is identical to Q.)*

Proof: Suppose, to the contrary, that there are no molecules P and Q, such that $P \sqsubseteq Q$, $\neg P \in \mathbf{S}$, and $\mathbf{S} \vdash Q$. We will show that then $\mathbf{S}$ must be satisfiable. Consider the following set of molecules:

$$\mathbf{D}(\mathbf{S}) \overset{\text{def}}{=} \{P \mid \ P \text{ is a submolecule of some molecule } Q \text{ such that } \mathbf{S} \vdash Q \ \}.$$

Section 9 shows that every H-structure, $\mathbf{H}$, has a corresponding F-structure, $\mathbf{I}_H$, such that $\mathbf{H} \models \mathbf{S}$ if and only if $\mathbf{I}_H \models \mathbf{S}$. Applying the construction from Section 9 to $\mathbf{D}(\mathbf{S})$ we obtain an F-structure, $\mathbf{M}$. Since $\mathbf{D}(\mathbf{S})$ is closed under deduction, it is easy to verify that $\mathbf{M}$ is, indeed, an F-structure. We claim that for every molecule P:

$$\mathbf{M} \models P \quad \text{if and only if} \quad P \in \mathbf{D}(\mathbf{S}) \tag{7}$$

The "if"-direction follows from soundness of the derivation rules. For the "only if"-direction, assume $\mathbf{M} \models P$ and consider the following two cases:

(i) *P is an is-a assertion or a predicate:*
If P is an is-a assertion, item (2) in the construction of $\mathbf{M}$ in Section 9 can be used to show that $\mathbf{M} \models P$ if and only if $P \in \mathbf{D}(\mathbf{S})$. If P is a predicate, item (9) can be used to show the same.

(ii) *P is an object molecule composed of atoms $\tau_1, \ldots, \tau_n$:*
Then $\mathbf{M} \models P$ if and only if $\mathbf{M} \models \tau_i$, $i = 1, \ldots, n$. By items (5), (6), (7), or (8) of Section 9 (depending on whether the method expression in τ_i is scalar or set-valued and whether it is a data or a signature expression), it follows that $\mathbf{M} \models \tau_i$ if and only if $\tau_i \in \mathbf{D}(\mathbf{S})$. Therefore, by the definition of $\mathbf{D}(\mathbf{S})$, there are molecules $Q_1, \ldots, Q_n$ deducible from $\mathbf{S}$, such that τ_i is a submolecule of Q_i, for $i = 1, \ldots, n$. Let Q be the canonical merge of $Q_1, \ldots, Q_n$. Then P is a submolecule of Q (since every constituent atom of P is also a constituent atom of Q) and Q is deducible from $\mathbf{S}$ (since $Q_1, \ldots, Q_n$ are deducible from $\mathbf{S}$ and Q is a merge of $Q_1, \ldots, Q_n$). Hence, P is in $\mathbf{D}(\mathbf{S})$, which proves (7).

By the definition of $\mathbf{D}(\mathbf{S})$, if $P \in \mathbf{S}$ is a positive literal then $P \in \mathbf{D}(\mathbf{S})$. Hence, by (7), $\mathbf{M} \models P$. For every negative literal $\neg P$ in $\mathbf{S}$, P is *not* a submolecule of any molecule deducible from $\mathbf{S}$, by the assumption made at the beginning of the proof. So, P is not in $\mathbf{D}(\mathbf{S})$. Again, by (7), $\mathbf{M} \not\models P$ and therefore $\mathbf{M} \models \neg P$. Thus, $\mathbf{M}$ satisfies every literal of $\mathbf{S}$, that is, it is a model for $\mathbf{S}$. $\Box$

Theorem 11.5 (Completeness of ground deduction) *If a set of ground clauses, $\mathbf{S}$, is unsatisfiable then there exists a refutation of $\mathbf{S}$.*

Proof: By Herbrand's Theorem, we can assume that $\mathbf{S}$ is finite. Suppose $\mathbf{S}$ is unsatisfiable. We will show that there is a refutation of $\mathbf{S}$ using a technique due to Anderson and Bledsoe [9]. The proof is carried out by induction on the parameter $excess(\mathbf{S})$, the number of "excess literals" in $\mathbf{S}$:

$$excess(\mathbf{S}) \overset{\text{def}}{=} (\text{ the number of occurrences of literals in } \mathbf{S}) - (\text{the number of clauses in } \mathbf{S}).$$

Basis: $excess(\mathbf{S}) = 0$. In this case, the number of clauses in $\mathbf{S}$ equals the number of occurrences of literals in $\mathbf{S}$. Hence either $\Box \in \mathbf{S}$ and we are done, or every clause in $\mathbf{S}$ is a literal. In the latter case, by Lemma 11.4, $\mathbf{S} \vdash \neg P$ and $\mathbf{S} \vdash Q$ for some molecules P, Q such that $P \sqsubseteq Q$. Applying the resolution rule to $\neg P$ and Q, we obtain the empty clause.

Induction Step: $excess(\mathbf{S}) = n > 0$. In this case, there must be a clause, C, in $\mathbf{S}$ that contains more than one literal. Let us distinguish this clause from other clauses and write $\mathbf{S} = \{C\} \cup \mathbf{S}'$, where $C = L \vee C'$ ($C' \neq \Box$ since we have assumed that C contains more than one literal). By the distributivity law, $\{L \vee C'\} \cup \mathbf{S}'$ is unsatisfiable if and only if so are $\mathbf{T}_1 = \{C'\} \cup \mathbf{S}'$ and $\mathbf{T}_2 = \{L\} \cup \mathbf{S}'$. Since $excess(\mathbf{T}_1) < n$ and $excess(\mathbf{T}_2) < n$, the induction hypothesis ensures that there are refutations of $\mathbf{T}_1$ and $\mathbf{T}_2$ separately. Therefore, $\mathbf{T}_1 \vdash \Box$, where $\Box$ is the empty clause. Let $dedseq_1$ denote the deduction that derives $\Box$ from $\mathbf{T}_1$. Applying the deductive steps in $dedseq_1$ to $\mathbf{S}$, we would derive either L or $\Box$. If $\Box$ is so produced, then S is refuted and we are done. Otherwise, if L is produced, it means that $\mathbf{S} \vdash L$. Let $dedseq_2$ denote the derivation that refutes $\mathbf{T}_2 = \{L\} \cup S'$ (which exists by the inductive assumption). Since $\mathbf{S}' \subset \mathbf{S}$ and $\mathbf{S} \vdash L$, it follows that if we apply $dedseq_1$ to $\mathbf{S}$ and then follow this up with steps from $dedseq_2$, we shall refute $\mathbf{S}$. $\Box$

Proposition 11.6 *There exists a unification algorithm that, given a pair of molecules T_1 and T_2, yields a complete set of mgu's of T_1 into T_2.*

Proof: The algorithm is given in Appendix C and its correctness is proved in Lemma C.1. $\Box$

Lemma 11.7 (Lifting Lemma) *Suppose C_1, C_2 are clauses and C_1', C_2' are their instances, respectively. If D' is derived from C_1' and C_2' (or from C_1' alone) using one of the derivation rules, then there exists a clause, D, such that*

- *D is derivable from the* factors *of C_1 and C_2 (resp., from C_1 alone) via a single derivation step;*
- *This derivation step uses the same inference rule as the one that derived D'; and*
- *D' is an instance of D.*

Proof: Consider each derivation rule separately. The proof in each case is similar to the corresponding proof in predicate calculus, since the notion of substitution is the same in both logics. Simple (but tedious) details are left as an exercise. $\Box$

Theorem 11.8 (Completeness of F-logic Inference System) *If a set $\mathbf{S}$ of clauses is unsatisfiable, then there is a refutation of $\mathbf{S}$.*

Proof: The proof is standard. Consider $\mathbf{S}^*$, the set of all ground instances of $\mathbf{S}$. By the ground case (Theorem 11.5), there is a refutation of $\mathbf{S}^*$. With the help of Lifting Lemma, this refutation can be then lifted to a refutation of $\mathbf{S}$. $\Box$

12 Data Modeling in F-logic

In this section, we define the notions of a logic program, a database, and a query, and illustrate the use of F-logic on a number of simple, yet non-trivial examples. We shall use the terms "deductive database" (or simply a database) and "logic program" interchangeably. As a first cut, we could say that a logic program in F-logic (abbr., an *F-program*) is an arbitrary set of F-formulae. However, as in classical logic programming, this definition is much too general and both pragmatic and semantic considerations call for various restrictions on the form of the allowed formulas.

12.1 Logic Programs and their Semantics

Perhaps the most popular class of logic programs is the class of *Horn programs.* A Horn F-program consists of *Horn rules*, which are statements of the form

$$head \longleftarrow body \tag{8}$$

where *head* is an F-molecule and *body* is a conjunction of F-molecules. Since (8) is a clause, this implies that all its variables are implicitly universally quantified.

Just as in classical theory of logic programs, it is easy to show that Horn programs have the *model-intersection property.* Thus, the intersection of all H-models of a Horn program, **P**, is also an H-model of **P**. This model is also the *least* H-model of **P** with respect to set-inclusion (recall that H-structures are sets of molecules).

By analogy with classical theory of Horn logic programs (see, *e.g.*, [72]), we can define an F-logic counterpart of the well-known $T_{\mathbf{P}}$ operator that, given a Horn F-program, **P**, maps H-structures of **P** to other H-structures of **P**. Given an H-structure, **I**, $T_{\mathbf{P}}(\mathbf{I})$ is defined as the smallest H-structure that contains the following set:[14]

$$\{head \mid head \longleftarrow l_1 \wedge \cdots \wedge l_n \text{ is a ground instance of a rule in } \mathbf{P} \text{ and } l_1, \ldots, l_n \in \mathbf{I}\}$$

Following a standard recipe, it is easy to prove that the least fixpoint of $T_{\mathbf{P}}$ coincides with the least H-model of **P** and that a ground F-molecule, φ, is in the least H-model of **P** if and only if $\mathbf{P} \models \varphi$.

Although Horn programs can be used for a large number of applications, their expressive power is limited. For more expressiveness, it is necessary to relax the restrictions on the form of the rules and allow negated F-molecules in *body* in (8). Such programs will be called *generalized* (or *normal*) F-programs.

For generalized programs, the elegant connection between fixpoints, minimal models, and logical entailment holds no more. In fact, such programs may have several minimal models and the "right" choice is not always obvious. However, it is generally accepted that the semantics of a generalized logic program is given by the set of its *canonic* models, which is a subset of the set of all minimal models of the program. Alas, in many cases there is no agreement as to which models deserve to be called canonic. Nevertheless, for a vast class of programs, called *locally stratified* programs, such an agreement has been reached, and it was shown in [89] that every such program has a unique canonic H-model.[15] For a locally stratified program, its unique canonic model goes under the name *perfect model.*

Appendix A develops a perfect-model semantics for locally stratified F-programs, which is an adaptation from [89]. For our current needs, however, we shall assume that *some* canonic H-model exists for each F-program under consideration; details of these models do not matter for the discussion that follows. Moreover, since most of our examples are based on Horn programs, the canonic model of each such program coincides with its unique minimal model.

Canonic Models and Equality

Equality has always been a thorny issue in logic programming because it does not easily succumb to efficient treatment. As a result, most logic programming systems—and all commercial ones—have the

[14]By itself, this set may not be an H-structure because of the closure properties that an H-structure must satisfy (see Section 9).

[15]We consider only *definite* rules, *i.e.*, rules that do not have disjunctions in the head.

so called "freeness axioms" built into them.[16] In practice, this boils down to a restriction that banishes equality predicates from the heads of program clauses.

Clearly, programming in F-logic cannot avoid such problems. Furthermore, equality plays a much more prominent role in object-oriented programming than in classical logic programming. In F-logic, for instance, equations can be generated implicitly, even if the program does not mention equality at all. For example, consider the following simple F-program:

$$john\,[father \rightarrow bob] \qquad\qquad john\,[father \rightarrow dad(john)] \tag{9}$$

Because *father* is a scalar attribute, this program entails $bob \doteq dad(john)$. Likewise, $car :: automobile$ and $automobile :: car$ together entail $car \doteq automobile$ because the class hierarchy must be acyclic.

Apart from the usual computational problems associated with the equality predicate, implicit generation of equations may be problematic from the practical point of view. Indeed, multiply-defined scalar methods, such as *father* above, or cycles in the class-hierarchy may be unintentional, a programmer's mistake. Therefore, it may be desirable to regard certain programs, such as (9) above, as inconsistent, unless $dad(john) \doteq bob$ is also defined explicitly. Of course, since the above program obviously has many models, "inconsistency" here should be taken to mean the absence of *canonic* models. Furthermore, the notion of canonic models needs special adjustment for equality, because many programs that generate implicit equality may well have canonic models in the usual sense. For instance, (9) certainly has a minimal H-model, which consists of the two given atoms, the equation $bob \doteq dad(john)$, plus all tautological F-atoms.

The canonic models that take special care of the equality will henceforth be called *equality-restricted canonical models*, which is defined next.

The first step is to split each program, $\mathbf{P}$, into two (not necessarily disjoint) subsets, $\mathbf{P}^{\doteq}$ and $\mathbf{P}^{rest}$. The *equality definition*, $\mathbf{P}^{\doteq}$, is the part that determines the equality theory of $\mathbf{P}$. The second part, $\mathbf{P}^{rest}$, specifies the rest.

Implicit here is the assumption that the programmer would specify which part of the program determines the "intended" equalities of the problem domain. Any other equality implied by $\mathbf{P}$ would then be interpreted as an inconsistency.

For simplicity, we shall assume that the equality definition, $\mathbf{P}^{\doteq}$, has only one canonic H-model. However, we do not assume any specific theory of canonic models. Let $\mathbf{H}^{\doteq}$ be the canonical H-model of $\mathbf{P}^{\doteq}$ and let EQ be the set of all non-trivial equations in $\mathbf{H}^{\doteq}$. (An equation, $s \doteq t$, is non-trivial if and only if s and t are not identical.)

Definition 12.1 (Equality-restricted Canonical Models) Let $\mathbf{M}$ be a canonical H-model of $\mathbf{P}$ (we do not assume that $\mathbf{M}$ is unique). We say that $\mathbf{M}$ is an *equality-restricted canonical model* of $\mathbf{P}$ if the set of nontrivial equations in $\mathbf{M}$ coincides with EQ. □

Coming back to our example, suppose that $bob \doteq dad(john)$ is not in the equality-defining part of (9). Then $\neg(bob \doteq dad(john))$ must hold in all canonic models of the program. However, since $bob \doteq dad(john)$ is a logical consequence of (9), this program has no equality-restricted canonic model.

[16] In logic programming, freeness axioms are also known as Clark's Equality Theory.

Queries

A *query* is a statement of the form $?-\ Q$, where Q is a molecule.[17] The set of *answers* to $?-\ Q$ with respect to an F-program, $\mathbf{P}$, is the smallest set of molecules that

- contains all instances of Q that are found in the canonic model of $\mathbf{P}$; and
- is closed under "$\models$".

The first condition is obvious and does not need further comments. The second condition is needed for the following reason. Suppose the database contains $john\,[children \twoheadrightarrow \{bob, sally\}]$ and the query is $?-\ john\,[children \twoheadrightarrow X]$. Then two instances of the query are implied by this database: $john\,[children \twoheadrightarrow bob]$ and $john\,[children \twoheadrightarrow sally]$. However, $john\,[children \twoheadrightarrow \{bob, sally\}]$ is not an instance of the query. So, without the closure with respect to $\models$, this molecule would not be part of the answer set, even though it is a logical consequence of the first two answers. On the other hand, this molecule is in the answer set of a logically equivalent query, $?-\ john\,[children \twoheadrightarrow \{X, Y\}]$.

Closure with respect to "$\models$" eliminates these anomalies and makes it easier to talk about query equivalence and containment without getting bogged down in minor syntactic differences that may occur in essentially similar queries.

The Structure of F-logic Programs

F-programs specify what each method is supposed to do, define method signatures, and organize objects along class hierarchies. Thus, every program can be split into three disjoint subsets, according to the type of information they specify:

- The *IS-A hierarchy declaration.* This part of the F-program consists of the rules whose head-literal is an is-a assertion.
- The *signature declaration.* This part contains rules whose head literal is an object-molecule that is built out of signature expressions only.
- The *object-base definition.* This part consists of rules whose head literals do not contain signatures or is-a expressions, *i.e.*, rules whose heads are either predicates or object-molecules built exclusively out of data expressions.

Note that the above classification considers rule-heads only. It is legal, therefore, for rules in one subsets to have body-literals defined in other subsets. In fact, as we shall see, certain applications may need such flexibility.

Informally, an object-base definition specifies what each method is supposed to do. Object definitions may be explicit, *i.e.*, given as facts, or implicit, *i.e.*, specified via deductive rules. Class-hierarchy declarations, as their name suggests, organize objects and classes into IS-A hierarchies. Signature declarations specify the types of the arguments for each method and the type of the output they produce.

[17] This does not limit generality, as every query can be reduced to this form by adding appropriate rules.

12.2 Examples of IS-A Hierarchies

Given an F-program and its canonic H-model, the IS-A hierarchy defined by the program is the set of all is-a atoms satisfied by the canonic model.

One simple example of a class-hierarchy declaration is given in Figures 2 and 3 of Section 3. In this hierarchy, *john* is a student and an employee at the same time; *phil* is an employee but not a student. The latter is *not* stated explicitly and *cannot* be derived using the normal logical implication, "$\models$", of Section 5.2. However, $\neg phil : student$ holds in the minimal model of the program of Section 3. Since the semantics is determined by this model, $\neg phil : student$ is considered to be a valid statement about the class hierarchy.

The idea of class hierarchies is an important ingredient in the phenomenon known as *inclusion polymorphism.* For instance, stating that students and employees are persons implies that all properties (*i.e.*, methods) applicable to persons must automatically be applicable to employees and students. In Figure 4 of Section 3, such properties are *name* (a scalar 0-ary method that returns objects of class *string*), *friends* (a set-valued attribute that returns objects of class *person*), and a few others. In F-logic, inclusion polymorphism is built into the semantics. It is manifested by a property discussed in Section 7.3, called *structural inheritance.*

Since classes are objects, they are represented by ground id-terms. This suggests that parametric families of classes can be represented via non-ground id-terms, which provides much of what is needed to support *parametric polymorphism.* For example, the following pair of clauses defines a parametric polymorphic class $list(T)$:

$$\begin{array}{l} nil : list(T) \\ cons(X,Y) : list(T) \longleftarrow X : T \wedge Y : list(T) \\ list(T) :: list(S) \longleftarrow T :: S \end{array} \tag{10}$$

Here $list(T)$ denotes a parametric family of classes of the form

$$\{\, list(t) \mid \text{where } t \text{ is a ground id-term} \,\}$$

and *cons* is the list constructor. For instance, if *int* denotes the class of integers then $list(int)$ is the class of lists of integers containing the elements *nil*, $cons(0, nil)$, $cons(0, cons(1, nil))$, and so on. The last clause above states that if T is a subclass of S then $list(T)$ is a subclass of $list(S)$. Note that this does not follow from the rest of the definition in (10) and has to be stated explicitly (if this property of lists is wanted, of course).

The above family of list-classes will be used later to define parameterized types for list-manipulation methods.

In F-logic, the IS-A hierarchy may depend on other data because is-a atoms can occur in rule heads. Classes defined in this way are analogous to views in relational databases. For instance,

$$Car : dieselCars(Year) \longleftarrow Car : car\,[engineType \rightarrow \text{``}diesel\text{''};\ makeYear \rightarrow Year]$$

defines a family of derived classes, parameterized by $Year$. Each class, is populated by diesel cars made in the appropriate year. For instance, $dieselCars(1990)$, would be populated by objects that represent diesel cars made in 1990.

In the previous example, population of the classes is determined by the properties of their objects. Sometimes, though, it desirable to have classes determined by the structure of the objects instead of by properties. For instance, the rule

$$X : merchandise \longleftarrow X : Y \wedge Y\, [price \Rightarrow (\,)]$$

defines a derived class, *merchandise*, that consists of all objects to which the attribute *price* applies. Note that an object would fall into the class *merchandise* if it has the attribute *price*, even if the price for that merchandise has not been set yet.

Observe that classes such as *merchandise*, *dieselCar(1990)*, or *list(int)*, are *virtual* in the sense that their *extensions* (*i.e.*, sets of members) are specified indirectly, via logical rules. It is not hard to see that virtual classes are akin to views in relational databases.

Our last example concerns the set-theoretic and the lattice-theoretic operators on the class hierarchy. F-logic does not require classes to form a lattice, *i.e.*, a pair of classes, c and c', does not have to have the lowest superclass or the greatest subclass. It is not even required that c and c' will have an intersection (or a union) class, *i.e.*, a class whose extension is the intersection (resp., union) of the extensions of c and c'. However, we can *define* class constructors that accomplish these tasks. For instance, the following rules define $and(X, Y)$ and $or(X, Y)$ to be intersection and union classes of its arguments, X and Y:

$$\begin{array}{lcl} I : or(X,Y) & \longleftarrow & I : X \\ I : or(X,Y) & \longleftarrow & I : Y \\ I : and(X,Y) & \longleftarrow & I : X \wedge\ I : Y \end{array}$$

Note that, in the canonical model, $and(X, Y)$ is not a subclass of X and Y, and neither $or(X, Y)$ is their superclass. The above rules relate only the *sets of members* of the two classes involved, not the classes themselves. If we also wanted to relate the classes, we would write:

$$\begin{array}{lcl} lsup(X,Y) :: C & \longleftarrow & X :: C \wedge Y :: C \\ X :: lsup(X,Y) & & \\ Y :: lsup(X,Y) & & \\ C :: gsub(X,Y) & \longleftarrow & C :: X \wedge\ C :: Y \\ gsub(X,Y) :: X & & \\ gsub(X,Y) :: Y & & \end{array}$$

Here, *lsup* is a constructor that defines the lowest superclass of X and Y, and *gsub* defines their greatest subclass. Note that, for instance, $gsub(X, Y)$ does *not* equal the intersection of the extensions of X and Y—it may be a strict subset of such an intersection. In other words, *gsub* and *lsup* construct lower and upper bounds in the class hierarchy, but not in the hierarchy of class extensions. However, by combining the rules for *and* and *gsub* (and for *or* and *lsup*) we can define constructors for lower and upper bounds in both hierarchies.

12.3 Examples of Type Declarations

Typing is a popular concept in programming languages. In its primitive form, it appears in traditional database systems under the disguise of schema declaration. In programming, typing can be very useful both as a debugging aid and as a means of maintaining data integrity. It allows the user to define correct usage of objects and then let the system detect ill-typed data and queries. The purpose of type declarations is to impose type-constraints on arguments of methods as well as on the results returned by the methods. For instance, in

$$\begin{array}{l} empl\,[salary \,@\, year \Rightarrow integer] \\ person\,[birthdate \Rightarrow year] \end{array}$$

the first molecule states that *salary* is a function that for any *empl*-object would return an object in the class *integer*, if invoked with an argument of class *year*. The second clause says that *birthdate* is an attribute that returns a year for any *person*-object. Section 13 discusses how typing constraints are imposed in F-logic. In the present section we shall only give some examples of type definitions, leaving the semantic considerations till later.

In the previous subsection, we have seen one example of a parametric family of classes, $list(T)$. A signature for this family can be given as follows:

$$list(T)[first \Rightarrow T;\ rest \Rightarrow list(T);\ length \Rightarrow int;\ append \,@\, list(T) \Rightarrow list(T)] \tag{11}$$

These signatures are parametric; they declare the attributes *first*, *rest*, *length*, and the method *append* as polymorphic functions that can take arguments of different type. As with any clause in a logic program, variables in (11) are universally quantified.

For instance, the signature for *append* says: if *append* is invoked on a $list(int)$-object with an argument from class $list(int)$, then the result must be an object of class $list(int)$. However, if the argument is, say, $list(real)$ then the output must be an object of class $list(real)$. This is because this invocations of *append* must conform to the signature $list(real)[append \,@\, list(real) \Rightarrow list(real)]$, which is an instance of (11). Note that the output of this invocation does not have to be in class $list(int)$, because the signature-instance $list(int)[append \,@\, list(int) \Rightarrow list(int)]$ does not cover the invocation in question (as the method *append* was invoked on a list of reals, not integers).

12.4 Examples of Object Bases

This section illustrates the expressive power of F-logic on a number of interesting and non-trivial examples, which include manipulation of sets and database schema, analogical reasoning, list processing, and others. In many cases, we shall omit signatures (usually when they are obvious or if they were discussed before). Also, since inheritance of properties will be discussed separately, all data expressions used in this subsection are of non-inheritable variety.

12.4.1 Set Manipulation

The ability to manipulate sets with ease is an important litmus test for an object-oriented data language. This subsection illustrates set-manipulation by expressing a number of popular operators, such as nesting and unnesting, set-comparison, and power-set operator. Other examples, such as set intersection and union operators can be found in [60].

Nesting and unnesting. Among the many set-related operations, the ability to restructure objects by nesting and unnesting are among the most important ones. Specifying these operations in F-logic is quite easy. Consider a class of objects with the following structure:

$$c\,[attr_1 \Rightarrow r_1;\ attr_2 \Rightarrow r_2]$$

One way to nest this class, say, on $attr_2$, is to specify another class, $nest(c, attr_2)$, with the signature

$$nest(c, attr_2)[attr_1 \Rightarrow r_1;\ attr_2 \Rightarrow\!\!\!\Rightarrow r_2]$$

This class is populated according to the following rule:

$$nested(Y) : nest(c, attr_2)[attr_1 \rightarrow Y;\ attr_2 \twoheadrightarrow Z] \quad \longleftarrow \quad X : c\,[attr_1 \rightarrow Y;\ attr_2 \rightarrow Z]$$

It is easy to see from either the semantics or the proof theory that this rule has the following meaning: to nest c on $attr_2$, populate the class $nest(c, attr_2)$ with the objects of the form $nested(y)$, such that their attribute $attr_2$ groups all z's that occur with y in some object x in class c.

Similarly, consider a class with the following signature:

$$c\,[attr_1 \Rightarrow r_1;\ attr_2 \Rightarrow\!\!\!\Rightarrow r_2]$$

To unnest this class on the attribute $attr_2$, we define another class, $unnest(c, attr_2)$, with the following signature:

$$unnest(c, attr_2)[attr_1 \Rightarrow r_1;\ attr_2 \Rightarrow r_2]$$

Identities of objects in this class depend on both the source-objects (from class c) and on the values returned by $attr_2$:

$$unnested(X, Z) : unnest(c, attr_2)[attr_1 \rightarrow Y;\ attr_2 \rightarrow Z] \quad \longleftarrow \quad X : c\,[attr_1 \rightarrow Y;\ attr_2 \twoheadrightarrow Z]$$

This rule says that unnesting of c on $attr_2$ involves specifying objects of the form $unnested(x, z)$, for each c-object, x, and for each value z of $attr_2$ on x. In the unnested objects, both attributes, $attr_1$ and $attr_2$, are scalar and are defined so as to flatten the structure of the original objects in class c.

Set comparison. Grouping is not only easy to express in F-logic, but it is also computationally more efficient than in some other languages, such as LDL [85], COL [2], or LPS [67]. The reason for this is that, in those languages, grouping is a second-order operation that requires stratification. We refer the reader to [60] for a more complete discussion.

However, this gain in efficiency has a price. For instance, in LDL, set-equality is easy to express because LDL treats sets as value-based entities and, in particular, LDL's equality operator is applicable to sets. In contrast, F-logic does not have a built-in equality operator for sets and each set is represented via an oid. Nevertheless, set-equality and containment can be defined via logical rules.

To see how, suppose we want to verify that the result returned by one set-valued attribute stands in a certain relation (*e.g.*, equals, contains, etc.) to the value of another attribute of some other object. Defining these relationships in F-logic is akin to comparing relations in classical logic programming:

$$\begin{array}{rcl}
notSubset(Obj, Attr, Obj', Attr') & \longleftarrow & Obj\,[Attr \twoheadrightarrow X] \wedge \neg Obj'[Attr' \twoheadrightarrow X] \\
subset(Obj, Attr, Obj', Attr') & \longleftarrow & \neg notSubset(Obj, Attr, Obj', Attr') \\
setUnequal((Obj, Attr, Obj', Attr') & \longleftarrow & notSubset(Obj, Attr, Obj', Attr') \\
setUnequal((Obj, Attr, Obj', Attr') & \longleftarrow & notSubset(Obj', Attr', Obj, Attr) \\
setEqual(Obj, Attr, Obj', Attr') & \longleftarrow & \neg setUnequal((Obj, Attr, Obj', Attr')
\end{array} \qquad (12)$$

It should be noted, however, that although expressing set-equality in F-logic is more cumbersome than in LDL, implementing the equality operator of LDL involves the use of negation as failure, too. Therefore, comparing sets in F-logic and LDL has the same complexity.

Data restructuring. The next example is an adaptation from [3]. The issue here is the representation of data functions of COL [2] and grouping of LDL [17].

Consider a relation, $graph(X, Y)$, whose tuples represent edges of a graph. The task is to re-structure this graph by representing it as a set of nodes, such that each node points to a set of its descendants. The corresponding F-program is very simple:

$$rebuiltGraph\,[nodes \twoheadrightarrow Node\,[descendants \twoheadrightarrow D]] \longleftarrow graph(Node, D)$$

where *rebuiltGraph* is an oid chosen to represent the object that describes the rebuilt graph. This rule also shows one more way to do nesting. This time, though, we are nesting a relation, *graph*, rather than a class.

The reader familiar with HiLog [34] may note that this rule can be made more general if we extend the syntax of F-logic to include variables that range over predicate names:

$$rebuilt(Rel)[nodes \twoheadrightarrow Node\,[descendants \twoheadrightarrow D]] \longleftarrow Rel(Node, D) \wedge Rel : relation$$

where *Rel* is a variable and *relation* is an appropriately defined class of binary relations. This rule will restructure any binary relation that is passed as a parameter. For instance,

$$?-\ rebuilt(yourFavouriteRel)[nodes \twoheadrightarrow Node\,[descendants \twoheadrightarrow D]]$$

will return a rebuilt version of *yourFavouriteRel*.

For another example, suppose that we have a method, *grade*, declared as follows:

$$course\,[grade\,@\,student \Rightarrow integer]$$

A method like this would be appropriate for the use by an administrator. However, students are likely to be denied access to this method in class *course* because this exposes the grades of other students. On the other hand, students may be entitled to see their own grades. To implement this policy, each student can be given access to the object that represents the achievements of that particular student. We can then define the method *grade* on *student*-objects as follows:

$$Stud\,[grade@Crs \rightarrow G] \longleftarrow Crs : course\,[grade@Stud \rightarrow G]$$

In this way, a student, say Mary, can access her grades by posing queries such as $?-\ mary\,[grade@vlsi \rightarrow G]$, but she cannot access grades by querying a *course*-object, *e.g.*, $?-\ vlsi\,[grade@mary \rightarrow G]$.

The power-set operator. Another interesting problem is to try and express the power-set operator, an operator that takes an arbitrary set and computes all of its subsets. First, we define the class of sets; it consists of objects with oid's of the form $add(a, add(\ldots))$. The intended meaning of a set-object denoted by $add(a, add(b, add(c, \emptyset)))$ is the set $\{a, b, c\}$. The class of sets is defined as follows:

$$\begin{array}{l} \emptyset : set \\ add(X, S) : set \;\longleftarrow\; S : set \\ add(X, add(Y, S)) \doteq add(Y, add(X, S)) \\ add(X, add(X, S)) \doteq add(X, S) \end{array} \qquad (13)$$

The first equation asserts that the order of adding elements to sets is immaterial; the last equation says that insertion of elements into sets is an idempotent operation.

Next, we define an attribute, *self*, that for every *set*-object returns the set of all members of this set:

$$\begin{array}{l} \emptyset\,[self \twoheadrightarrow \{\,\}] \\ add(X,L)[self \twoheadrightarrow X] \longleftarrow L : set \\ add(X,L)[self \twoheadrightarrow Y] \longleftarrow L : set\,[self \twoheadrightarrow Y] \end{array} \tag{14}$$

Finally, the *powerset* method is defined as follows:

$$\begin{array}{lcl} S\,[\,powerset \twoheadrightarrow \emptyset\,] & \longleftarrow & S : set \\ S\,[\,powerset \twoheadrightarrow add(Z,L)\,] & \longleftarrow & S : set\,[self \twoheadrightarrow Z;\ powerset \twoheadrightarrow L] \\ & & \wedge\ L : set\ \wedge\ \neg L\,[self \twoheadrightarrow Z] \end{array} \tag{15}$$

The first clause says that the empty set is a member of any power-set. The second rule says that a set obtained from a subset, L, of S by adding an element $Z \in S - L$, is a subset of S and, thus, is also a member of the power-set of S.

12.4.2 Querying Database Schema

The higher-order syntax of F-logic makes it possible to query and manipulate certain meta-information about the database, such as its schema. Schema querying was also discussed in [63] and recently in [34]. However, the treatment in [63] is not as general and integrated as in F-logic, while [34] is a relational rather than an object-oriented language.

Typically, database queries are specified with respect to a fixed, known scheme. Experience shows, however, that this assumption is unrealistic and ad hoc schema exploration may often be necessary. This means that the user has to apply intuitive or exploratory search through the structure of the scheme and the database at the same time and even in the same query (cf. [84]). Many user interfaces to commercial databases support browsing to some extent. The purpose of the following examples is to demonstrate that F-logic provides a unifying framework for data *and* schema exploration. Once again, we refer to the example in Section 3.

For each object in class *faculty*, the following pair of rules collects all attributes that have a value in class *person*:

$$\begin{array}{lcl} interestingAttributes(X)\,[attributes \twoheadrightarrow L] & \longleftarrow & X : faculty[L \rightarrow Z : person] \\ interestingAttributes(X)\,[attributes \twoheadrightarrow L] & \longleftarrow & X : faculty\,[L \twoheadrightarrow Z : person] \end{array} \tag{16}$$

For every *faculty*-object, *o*, these rules define another object, *interestingAttributes*(*o*), with a set-valued attribute, *attributes*. The intuitive reading of (16) is: If L is an attribute defined on an object, X, and if L has a *person*-value, Z, then L must belong to the result that *attributes* has on the object *interestingAttributes*(X).

Thus, in the example in Section 3, we would obtain: *interestingAttributes*(*bob*) = {*boss*} and *interestingAttributes*(*mary*) = {*friends*}. Deleting the restriction *person* in (16) would add those attributes that have any value on X, not just a *person*-object. In that case, *interestingAttributes*(*bob*) will also contain *name*, *age*, and *affiliation*, while *interestingAttributes*(*mary*) will include *name*, *highestDegree*, and *affiliation*.

Another interesting problem arises when one needs to obtain all objects that reference some other object directly or indirectly (via subobjects). The method *find*, below, returns the set of all such objects that reference *Stuff*:

$$\begin{aligned}
&browser\,[find\,@\,Stuff \twoheadrightarrow X] \longleftarrow X\,[Y \rightarrow Stuff]\\
&browser\,[find\,@\,Stuff \twoheadrightarrow X] \longleftarrow X\,[Y \twoheadrightarrow Stuff]\\
&browser\,[find\,@\,Stuff \twoheadrightarrow X] \longleftarrow X\,[Y \rightarrow Z] \wedge browser\,[find\,@\,Stuff \twoheadrightarrow Z]\\
&browser\,[find\,@\,Stuff \twoheadrightarrow X] \longleftarrow X\,[Y \twoheadrightarrow Z] \wedge browser\,[find\,@\,Stuff \twoheadrightarrow Z]
\end{aligned}$$

For the example in Section 3, the query $?-\ browser\,[find\,@\,\text{“}CS\text{”} \twoheadrightarrow X]$ would return the set $\{cs_1, cs_2, bob, mary\}$.

12.4.3 Representation of Analogies

Reasoning by analogy is an active field of research (*e.g.*, see a survey in [47]). Apart from the semantic and heuristic issues, finding suitable languages in which to specify analogies is also a challenge. This subsection shows how certain kinds of analogies can be specified in F-logic.

In Section 3, we defined a method that, when applied to a host-object of type *person* with a *person*-argument, returns the set of all joint works of the two persons involved. This method is just one instance of a general situation where joint projects, common hobbies, and other commonalities may be involved. We can abstract this concept and define a "generic" method for joint things:

$$\begin{aligned}
X\,[joint(M)\,@\,nil \twoheadrightarrow Z] &\quad\longleftarrow\quad X\,[M \twoheadrightarrow Z]\\
X\,[joint(M)\,@\,cons(Obj, Rest) \twoheadrightarrow Z] &\quad\longleftarrow\quad Obj\,[M \twoheadrightarrow Z] \wedge X\,[joint(M)\,@\,Rest \twoheadrightarrow Z]
\end{aligned}$$

The first rule here describes things that X has in common with an empty list of objects, *nil*. The second rule, then, says that X has object Z in common with each member of the list $cons(Obj, Rest)$ if Obj has Z and Z is a common object that X shares with the rest of the list.

Our second example describes similarities among classes, specified via the *like*-relationship (cf. [48, 12]). For instance, one can say, "The concept of a Whale is *like* the concept of a Fish via habitat" or, "A Pig-Like-Person is *like* a Pig via nose and legs."

Like-similarity can be expressed by means of a ternary predicate, *like*. For instance, in $like(pig\text{-}like, pig, cons(legs, cons(nose, nil)))$, the third argument would list the attributes that relate the first two arguments. We could then define:

$$C_1[Attr \rightarrow P] \longleftarrow C_2[Attr \rightarrow P] \wedge like(C_1, C_2, PropList) \wedge member(Attr, PropList)$$

where *member* is a membership predicate that can be defined by a standard Prolog program.

The above technique works well for specifying similarity via *attributes*, *i.e.*, via 0-ary methods. To represent similarity via methods of arbitrary arities we would need one $like_n$ predicate, for each arity $n \geq 0$, and one rule of the form

$$\begin{aligned}
C_1[M\,@\,X_1, \ldots, X_n \rightarrow P] \longleftarrow C_2[M\,@\,X_1, \ldots, X_n \rightarrow P] \quad &\wedge \quad like_n(C_1, C_2, MethdList)\\
&\wedge \quad member(M, MethdList)
\end{aligned}$$

12.4.4 List Manipulation

The following is an F-logic counterpart of a canonic textbook example:

$$\begin{aligned}
&nil\,[append\,@\,L \rightarrow L] \longleftarrow L : list(T)\\
&cons(X, L)[append\,@\,M \rightarrow cons(X, N)] \quad \longleftarrow L : list(T)[append\,@\,M \rightarrow N] \wedge X : T
\end{aligned} \tag{17}$$

Here L, M, and N are list-objects, while *append* is a method that applies to list-objects; when invoked with a list-object, b, as an argument on a list-object, a, as a host, *append* returns a concatenation of a and b. A parametric family of list-classes, $list(T)$, and their signatures were defined earlier, in (10) of Section 12.2 and in (11) of Section 12.3, respectively.

12.4.5 A Relational Algebra Interpreter

Writing an intelligible Prolog interpreter for relational algebra is a somewhat challenging task. First, one should parse the relational expression given as input. For example, an expression $((P \times V) \cup W) \bowtie_{cond} (Q - R)$ would be converted into a term of the form

$$join(\ union(prod(P, V),\ W),\ minus(Q, R),\ cond\ )$$

This part of the job is quite standard and can be found in many guides to Prolog. Once the expression is parsed, it has to be evaluated. Surprisingly, evaluators for joins, Cartesian products, and the like are rather cumbersome and may take a couple of pages of Prolog code. The main difficulty here is the fact that arities of the expressions to be joined are unknown at compile time, which calls for the use of various nonlogical features such as "=..", "arg", and "functor". We shall see that a program for the relational join evaluator can be written in just three rules in F-logic.

Relations are represented as in Section 6, that is, each relation, p, is a class of tuple-objects. Given a pair of relations, p and q, and a join condition, *cond*, the program defines a relation-object represented via an oid of the form $join(p, q, cond)$. Join-conditions (which are assumed to be equalities) are represented as lists of pairs of the form $cond(A, B, cond(\ldots))$. For instance, $cond(A_1, B_1, cond(A_2, B_2, nil))$ encodes the equi-join condition $A_1 = B_1 \wedge A_2 = B_2$.

The first deductive rule, below, says that if i and j are oid's of tuples in relations p and q, respectively, then $new(i, j)$ is the oid of a tuple in a *nil*-join of p and q (*i.e.*, in a Cartesian product of p and q). The second rule recursively computes the oid's of all tuple-objects in the equi-join. These tuple-objects are not fully defined, however, as we have not yet specified the values of their attributes. The third rule, therefore is needed to extract these values from the source-tuples i and j:

$$\begin{array}{rcl} new(I, J) : join(P, Q, nil) & \longleftarrow & I : P \wedge\ J : Q \\ new(I, J) : join(P, Q, cond(L, M, Rest)) & \longleftarrow & I : P\,[L \rightarrow X] \wedge J : Q\,[M \rightarrow X] \\ & & \wedge\, new(I, J) : join(P, Q, Rest) \\ new(I, J)[L \rightarrow X, rename(M) \rightarrow Y] & \longleftarrow & I : P\,[L \rightarrow X] \wedge J : Q\,[M \rightarrow Y] \end{array} \quad (18)$$

Note that in the last rule, the attributes coming from the second relation in the join are renamed in order to avoid name clashes. When relations do not have common attributes or when we are interested in natural joins rather than equi-joins, this renaming would not be needed and so the symbol *rename* in the last rule could be dropped. Additionally, for natural joins, the join-condition itself can be simplified into a list of attributes instead of the list of pairs used above.

Now, if *dept* has attributes *dname* and *mngr*, while the relation *empl* has attributes *dname* and *ename*, the query $?-\ X : join(dept, empl, cond(dname, dname, nil))[Y \rightarrow Z]$ will return a join of the two relations on the attribute *dname*.

Actually, the above program does a more general job since it joins *classes* rather than just relations. It also demonstrates a difference that syntax can make: In Prolog, a purely logical specification of a join entails a rather sophisticated encoding of relations, which—in the end—results in a code that can hardly be called declarative.

13 Well-Typed Programs and Type Errors

In a strongly typed language, a method can be invoked only when the invocation is allowed by one of the signatures specified for this method.[18] For instance, in the relational model, one can use only those attributes that are defined for the particular relation at hand; attempting to use an attribute that is not defined for this relation will normally result in an error.

To illustrate, we reproduce an example from Section 12.3:

$$\begin{array}{l} empl :: person \\ empl\,[salary \,@\, year \Rightarrow integer] \\ person\,[birthdate \Rightarrow year] \end{array}$$

Here we would like to make it illegal for an *empl*-object to call methods other than *birthdate* and *salary*. Similarly, for persons that are not employees, we would like to prohibit the use of the attribute *salary* altogether, since it is not covered by a suitable signature in class *person*. In other words, we would like to say that (in the above example) the signature of *person*-objects consists precisely of one method, *birthdate*, and that the signature of *empl*-objects has precisely two methods, *salary* and *birthdate*. Both methods are scalar; *birthdate* expects no arguments, while salary needs one—a *year*-object.

Unfortunately, by itself these declarations *do not* ensure that the corresponding typing constraints hold in a canonic H-model, because connection is missing between signatures and data expressions. This missing link is now provided in the form of the well-typing conditions:

Definition 13.1 (Typed H-structures—Non-inheritable Data Expressions) Let $\mathbf{I}$ be an H-structure. Suppose α is a non-inheritable data atom of the form $o\,[m \,@\, a_1, \ldots, a_k \rightsquigarrow v] \in \mathbf{I}$ and β is a signature atom of the form $d\,[m \,@\, b_1, \ldots, b_k \approx\!> \cdots]$. We shall say that β *covers* α if, for each $i = 1, \ldots, k$, we have $o : d$, $a_i : b_i \in \mathbf{I}$. Here $\rightsquigarrow$ and $\approx\!>$ stand, respectively, for $\rightarrow$ and $\Rightarrow$ or for $\twoheadrightarrow$ and $\Rightarrow\!\!>$.

We shall say that $\mathbf{I}$ is a *typed* H-structure *with respect to non-inheritable* data expressions if the following conditions hold:

- Every non-inheritable data atom in $\mathbf{I}$ is covered by a signature atom in $\mathbf{I}$; and
- If a non-inheritable data atom, $o\,[m \,@\, a_1, \ldots, a_k \rightsquigarrow v] \in \mathbf{I}$, is covered by a signature of the form $d\,[m \,@\, b_1, \ldots, b_n \approx\!> w] \in \mathbf{I}$, where $\rightsquigarrow$ and $\approx\!>$ are as before, then $v : w \in \mathbf{I}$. □

Note that the first condition is a restriction on the domain of the definition of methods, and on when a method can be invoked as a scalar or a set-valued method. In plain terms, it says that every non-inheritable expression in $\mathbf{I}$ must be covered by a signature. The second condition says that every non-inheritable expression must satisfy all constraints imposed by the signatures that cover that expression.

When it comes to typing, inheritable expressions are *not* fully analogous to non-inheritable expressions, despite their semantic and proof-theoretic similarities. This is because inheritable expressions are *used* differently. Indeed, consider $highestDegree \bullet\!\!\rightarrow phd$, a property of class *faculty* defined in Section 3. This expression is supposed to be inherited by the members of class *faculty* and, thus, it is also a property of these members. Therefore, it must obey all typing constraints imposed on *faculty*. This idea is formalized in Definition 13.2, below.

[18]By an *invocation* we mean a tuple of the form $\langle obj, \overrightarrow{args} \rangle$, where *obj* is the host-object of the invocation and $\overrightarrow{args}$ is a list of arguments of the invocation.

It is instructive to compare the above expression to a monotonic non-inheritable property of *faculty*, $avgSalary \rightarrow 50000$. Since this expression applies not to the members of class *faculty* but to the object *faculty* itself, it does not have to comply with signatures specified for *faculty* as a class. Instead, Definition 13.1 applies here, which means that there should be a signature defined for some class that contains *faculty* as a member and that covers the invocation $faculty\,[avgSalary \rightarrow 50000]$.[19]

Definition 13.2 (Typed H-structures—Inheritable Data Expressions) Let **I** be an H-structure. Suppose α is an inheritable data atom of the form $c\,[m \,@\, a_1, \ldots, a_k \bullet\!\!\rightsquigarrow v] \in \mathbf{I}$ and β is a signature atom of the form $d\,[m \,@\, b_1, \ldots, b_k \approx\!> \cdots]$, where $\bullet\!\!\rightsquigarrow$ and $\approx\!>$ stand, respectively, for $\bullet\!\!\rightarrow$ and $\Rightarrow$ or for $\bullet\!\!\twoheadrightarrow$ and $\Rightarrow\!\!\!>$. We shall say that β *covers* α if $c :: d$, $a_i : b_i \in \mathbf{I}$.

An H-structure, **I**, is said to be *typed with respect to inheritable* data expressions if the following conditions hold:

- Every inheritable data atom in **I** is covered by some signature in **I**; and
- If an inheritable data atom, $c\,[m \,@\, a_1, \ldots, a_k \bullet\!\!\rightsquigarrow v] \in \mathbf{I}$, is covered by a signature of the form $d\,[m \,@\, b_1, \ldots, b_n \approx\!> w] \in \mathbf{I}$, where $\bullet\!\!\rightsquigarrow$ and $\approx\!>$ are as before, then $v : w \in \mathbf{I}$. □

Note that the only essential difference between Definitions 13.2 and 13.1 is in the way "coverage" is defined. In Definition 13.1, the atom $o[\ldots]$ is covered by $d[\ldots]$ if, among other things, $o : d$ holds. In contrast, in Definition 13.2, $c[\ldots]$ is covered by $d[\ldots]$ if $c :: d$. That is, in the first case, class membership is required, while in the second case it must be subclassing. In other words, for object's non-inheritable properties, signatures that matter are those that are specified for classes where the object is a *member*. In contrast, for inheritable properties, signatures that count are those that come from classes where the object is a *subclass*.

Definition 13.3 (Typed H-Structures—All Data Expressions) The four conditions in Definitions 13.1 and 13.2 are called the *well-typing conditions*. An H-structure, **I**, is *typed* if all well-typing conditions are satisfied, *i.e.*, if **I** is typed with respect to inheritable and non-inheritable data expressions. □

We are now ready to define the notion of well-typed F-programs:

Definition 13.4 (Typed Canonic Models) A *typed canonic model* of **P** is a model for **P** that is a typed H-structure *and* also a canonic model of **P** in the usual sense.[20] □

Definition 13.5 (Well-typed Programs) An F-program **P** is *well-typed* (or is *type-correct*) if every canonic H-model of **P** is a typed canonic H-model. Otherwise, **P** is said to be *ill-typed*. □

In plain words, our notion of type-correctness amounts to the following: A typed F-structure is one where all data atoms comply with all relevant signatures, *i.e.*, with all signatures that cover them. Therefore, saying that *all* canonic F-models of a program must be typed is tantamount to saying that the program does not imply data atoms that do not comply with the signatures implied by that program. In [33], this property is called *semantic adequacy*.

[19] Such a class, called *employmentGroup*, was introduced towards the end of Section 3.

[20] As determined by the chosen theory of canonic models, such as the theory of perfect F-models described in Appendix A.

To illustrate, consider the following F-program, **P**:

$$\begin{array}{llll} bob : faculty & bob : manager & 1989 : year & manager :: empl \\ mary : faculty & john : manager & 10000 : int & faculty :: empl \end{array}$$

$$\begin{array}{l} mary\,[boss \to bob] \\ empl\,[boss \Rightarrow manager;\ salary\,@\,year \Rightarrow integer] \\ faculty\,[boss \Rightarrow faculty] \end{array}$$

The minimal H-model of **P** will contain the following signature for *faculty*, which is inherited from *empl*:

$$faculty\,[boss \Rightarrow\ (faculty, manager);\ \ salary\,@\,year \Rightarrow integer]$$

It will also have other obvious atoms, such as $mary : empl$ and $john : empl$. Clearly, this model satisfies the well-typing conditions of Definition 13.3 and, therefore, it is a typed F-structure. Hence, **P** is well-typed.

Suppose now that **P** had $mary\,[boss \to john]$ instead of $mary\,[boss \to bob]$. Then **P** would have had a type error because the atom $mary\,[boss \to john]$ clashes with the typing $empl\,[boss \Rightarrow faculty]$. Indeed, $john$ is not known to be a faculty and hence $\neg john : faculty$ holds in the minimal model of **P**. Likewise, the clause $mary\,[salary \to 10000]$ would have caused a type error, because no signature was specified for the 0-ary version of the method *salary* (only the unary version of *salary* has a signature in **P**), so $\neg faculty\,[salary \Rightarrow\ (\,)]$ and $\neg empl\,[salary \Rightarrow\ (\,)]$ must hold in the minimal model of **P**. But this, together with $mary\,[salary \to 10000]$, defeats the first well-typing condition of Definition 13.3.

For another example, consider:

$$\begin{array}{l} ibm : company \\ john\,[affiliation \to ibm] \\ X : employee \longleftarrow X\,[affiliation \to I] \\ X\,[affiliation \Rightarrow company] \longleftarrow X : employee \end{array} \tag{19}$$

This F-program is well-typed because its minimal model is also a typed H-model. In contrast, if the last rule in (19) were replaced by

$$X\,[name \Rightarrow string] \longleftarrow X : employee$$

then the program would cease to be well-typed because the attribute *affiliation* would then have no declared signature and, hence, $\neg c\,[affiliation \Rightarrow\ (\,)]$ would hold in that model, for every class c. This, together with $john\,[affiliation \to ibm]$, defeats the first well-typing requirement for typed H-structures (Definition 13.1). Therefore, even though the modified program has a minimal model, it is not a typed model, so the program is ill-typed.

Discussion

Definition 13.5 provides semantic grounds for developing algorithms for static type-checking and supplies a yardstick for verifying their correctness. It also provides a uniform framework for comparing various such algorithms.

To better see the role of type correctness in the overall schema of things, observe that this notion is not part of F-logic *per se* but, rather, it belongs to the meta-theory of the logic. As such, neither the semantics nor the proof theory depends on the particular way well-typing is defined. Even the sublogic of signatures is independent of Definition 13.5. Therefore, in principle, other definitions of type-correctness can be adapted for use in conjunction with F-logic.

The need for a new notion of type-correctness in F-logic arose because existing theories are not sufficiently general to adequately type many kinds of F-programs (although some could be adapted for restricted classes of F-programs, *e.g.*, [49]). A semantically rich logical theory, like F-logic, calls for a semantical, model-based notion of type-correctness—a notion that would apply to all meaningful untyped programs and one that would be compatible with the informal meaning usually associated with correct typing (instead of being a by-product of a concrete but ad hoc type-verification procedure). Such a general notion was sorely missing from the literature.

On the other hand, while Definition 13.5 accommodates a vast class of programs, it is weaker than one may like it to be. As noted earlier, type-correctness in F-logic amounts to saying that well-typed programs are guaranteed to not imply facts that are inconsistent with signatures specified for these programs. Clearly, this is a minimum one should require of a correct logic program. However, another source of type errors comes from rules that never fire (because some body literal is false) and that have signature-incompatible literals in the body. Unfortunately, this latter kind of errors is not captured by Definition 13.5. In the terminology of [33], this means that the above notion of type correctness is *syntactically inadequate*. A detailed discussion of these issues and some solutions can be found in [108, 33].

Because the above notion of well-typing is so general, it comes as no surprise that it is undecidable to check if an arbitrary program is well-typed [59]. However, one can devise algorithms that would be sufficient for various special classes of programs. Such algorithms is a topic for further research; for classical logic programs, a similarly defined notion of type-correctness was studied in [108], both semantically and algorithmically.

To give the reader a better idea about the envisioned use of the F-logic type system, we briefly mention the concept of *type inference*. Originally, the notion of type-correctness was applied to strongly typed languages, languages that require the programmer to supply *complete* type specification with each program. However, in logic programming, complete type specification is often seen as a burdensome requirement. Adapting the ideas from ML to Prolog, Mishra [81] proposed to use type inference for determining the implied type for each predicate in a program and then use the inferred types for trouble-shooting. For a program, **P**, its inferred type can be taken to mean the "strongest"[21] type declaration under which **P** is well-typed.

While strong-typing may be too taxing for logic-based languages, pure type-inferencing would be another extreme. Indeed, in the absence of additional constraints, *some* type can always be inferred for *any* program. This inferred type can therefore be used only as an advice to the programmer, something that may or may not trigger an alarm when the user feels that the inferred type does not match the intuition. Therefore, we believe—as was also suggested by others (*e.g.*, [111, 109])—that type-inference combined with type checking is a suitable compromise. Roughly, the idea is to use type inference only for methods with *no* explicitly declared signatures. The types inferred for such methods are then presented to the user for inspection. If the inferred types seem wrong, the user may try to find an error in the program. On the other hand, methods *with* explicit signatures will be type-checked by the system using the semantics given earlier in this section. Details of this approach are sketched in [59, 108].

[21]In a sense that will not be discussed here—see [59].

14 Encapsulation

Encapsulation is a major concept in the suite of notions comprising the object-oriented paradigm. In a nutshell, encapsulation is a software-engineering technique that requires special designations for each method in a class. Methods designated as *public* can be used to define other methods in other classes. In contrast, methods designated as *private* can be used only in the class where they are declared. Other methods may be exported to some specific classes, but not to all classes.

While the idea of encapsulation is simple, its logical rendition is not. In [79], Miller suggested a way to represent modules in logic programming via the intuitionistic embedded implication. Chen [32] defines modules as second-order objects, where data encapsulation inside modules is represented using existential quantifiers. In their present form, these approaches are not sufficiently general for use in F-logic. It would be interesting to see, though, if these approaches can be extended to make them suitable for method encapsulation in an object-oriented logic. We should also mention the recent work by Bugliesi and Jamil [26]. While their language and the notion of encapsulation are more limited than ours, encapsulation is made part of the syntax, semantics, and the proof theory, which could be a promising direction in this area.

In contrast to [32, 79, 26], we view various encapsulation mechanisms as no more than type-correctness policies, albeit somewhat more elaborate than usual. This approach is quite general and it can model a wide variety of encapsulation mechanisms. Its other advantage is that it does not require extensions to the logic. Instead, encapsulation is treated as a type-correctness discipline and, thus, it is a *meta-logical* notion in our approach.[22]

The general outline of our encapsulation-as-type-correctness approach is as follows. First, all program clauses are grouped into *modules*. Then each module is checked for type-correctness separately. The notion of type-correctness we have in mind here is somewhat more elaborate than in Section 13. The main difference is that the set of signatures that will be used in determining type-correctness of any given module will now include not only the signatures specified for that module but also signatures exported to this module by other modules.[23]

It should be noted that in most object-oriented systems, encapsulation policies are centered around the concept of a class, not module. However, in a language where classes can be virtual, *i.e.*, defined via logical clauses, encapsulating each class seems to be not feasible. Nevertheless, modules can be made to fit exactly one class definition and, thus, they provide a way to encapsulate any *fixed* number of classes (which is more than what current systems are capable of doing anyway).

Traditionally, object-oriented systems were using the idea of encapsulation for hiding implementation details, and our approach is no different in this respect. However, a combination of modules with type-correctness can serve another useful purpose: authorization-based control over access to objects. This can be achieved along the following lines. First, we associate a module with each user so that every rule or query that the user adds to the system is automatically made part of that user's module. This simple scheme can control unauthorized access, because any user query would constitute a type error (and will be rejected) if it were to invoke a method that is not exported into that user's module.

[22] An interesting feature of [79, 32] is that they achieve encapsulation by purely logical means.

[23] Public signatures are considered to be exported to every module.

14.1 An Example

To illustrate the above ideas, consider a program consisting of the following three modules:

$$\begin{array}{l}
\textbf{module}\ \ \textbf{person}\ \{ \\
\quad person\,[\ \ \textbf{public}:\ birthyear \Rightarrow year;\ age \Rightarrow int;\ name \Rightarrow string\,] \\
\quad P\,[\,age \to Y' - Y\,]\ \ \longleftarrow\ \ P : person\,[\,birthyear \to Y\,] \wedge thisyear() = Y' \\
\quad \%\ \ thisyear()\ \text{returns the current year.} \\
\}
\end{array} \tag{20}$$

In this module, all attributes are said to be public, so they are readily available to other modules.

$$\begin{array}{l}
\textbf{module}\ \ \textbf{faculty}\ \{ \\
\quad faculty\,[\ \ \textbf{public}:\ funding@project \Rightarrow int;\ projects \Rightarrow\!\!\!\Rightarrow project;\ affiliation \Rightarrow dept] \\
\quad project\,[\ \ \textbf{public}:\ title \Rightarrow string;\ principal \Rightarrow faculty \\
\qquad\qquad\ \ \textbf{export-to(accounting):}\ \ account \Rightarrow account;\ budget \Rightarrow budget\,] \\
\quad F\,[funding@P \to X]\ \ \longleftarrow\ \ F : faculty\,[projects \twoheadrightarrow P\,[budget \to B\,[total \to X]\,]\,] \\
\quad ?-\ \ F : faculty\,[\,projects \twoheadrightarrow P\,[account \to A\,[balance \to X]\,]\,] \\
\}
\end{array} \tag{21}$$

The above represents a fragment of module **faculty**. The methods *account* and *budget* are not visible in modules other than **faculty** and the module **accounting** to which they are exported. In this particular case, the reason for shielding *account* and *budget* is that of protection rather than encapsulation—we assume that project expenditures are not in public domain. However, the total amount of funds is made available through a public method, *funding*, accessible via *faculty*-objects. Also, inside the module **faculty**, balance of each project is readily available through the public method *balance* of class *account* (defined next). Therefore, the last clause in (21) does not break encapsulation.

$$\begin{array}{l}
\textbf{module}\ \ \textbf{accounting}\ \{ \\
\qquad budget\,[\ \ \textbf{public}:\ \ \ salaries \Rightarrow int;\ equipment \Rightarrow int;\ supplies \Rightarrow int;\ total \Rightarrow int\,] \\
\quad\ account\,[\ \ \textbf{private}:\ \ project \Rightarrow project; \\
\qquad\qquad\quad\ \ \textbf{public}:\ \ \ expended \Rightarrow int;\ encumbered \Rightarrow int;\ balance \Rightarrow int\,] \\
\quad B\,[\,total \to X + Y + Z\,]\ \ \longleftarrow\ \ B : budget\,[\,salaries \to X;\ equipment \to Y;\ supplies \to Z\,] \\
\quad A\,[\,committed \to X + Y]\ \ \longleftarrow\ \ A : account\,[\,expended \to X;\ encumbered \to Y\,] \\
\quad A\,[\,balance \to X - Y - Z\,]\ \ \longleftarrow\ \ A : account\,[\,project \to P[budget \to B\,[total \to X]];\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad expended \to Y;\ encumbered \to Z\,] \\
\quad ?-\ \ bluff : project\,[account \to A[balance \to X]\,] \\
\}
\end{array} \tag{22}$$

Unlike module **person**, the last two modules, **faculty** and **accounting**, are not built around any specific class. Instead, they serve as depositories for method definitions and as authorization domains for members of the faculty and for accountants. Note that **faculty** exports the methods *budget* and *account* to module **accounting**, which makes it legal to invoke these methods in the last two clauses in **accounting**. Also, although most of the information about accounts is made public in module **accounting**, the information identifying projects associated with each account is not. This is achieved by making *project* a private attribute in class *account*.

In the above example, the construct **module** { $\cdots$ } and the keywords **private**, **public**, and **export-to** are not part of the logic—they are *meta-annotations* that will be used to modify the notion of type correctness presented in Section 13 (which itself was presented as a meta-logical concept). The following section gives a formal account of type-correctness for programs with a module structure.

14.2 Modules and Type Correctness

The above example contains all major elements of our treatment of encapsulation. The key idea is to use a more elaborate notion of type-correctness to preclude illegal method invocations. This requires imposition of a *module structure* on each F-program, which is specified via meta-logical annotations. The new notion of type correctness builds upon the notion introduced in Section 13; it reduces to the old notion when the entire F-program is viewed as one big module.

Definition 14.1 (Module Structures) Let **P** be an F-program. A *module structure* of **P** consists of the following:

(i) A decomposition of the set of clauses in **P** into a collection of named (not necessarily disjoint) subsets, called *modules*;

(ii) Annotations (**private**, **public**, or **export-to**(module)) attached to each signature expression that occurs in the head of a rule in **P**. □

Since modules do not necessarily consist of disjoint sets of clauses, one module may be a sub-module of another module. Note further that, as illustrated above, some modules may represent authorization domains for various classes of users. Clauses and queries introduced by these users are assumed to enter the appropriate modules automatically. Also, Definition 14.1 does not assume that there is a correspondence between modules and classes; however, if desired, modules can be made to fit exactly one class, for any class in a given finite set. Finally, it is important to keep in mind that modules do not affect canonical models—only the notion of type correctness is affected (see below).

To be of practical use, the above definition of module structures needs further specialization. One thing that immediately comes to mind is that certain module structures may have internal contradictions. For instance, a method, m, may be declared as public in one module and as private in another. We view a module structure as *incoherent* if more than one signature of m with different annotations covers the same data atom.

Another situation where a module structure may be viewed as incoherent is when the same method expression is declared as private in two separate modules. If this were allowed, any module could import any signature by declaring it private, and so we would loose control over what can be *exported* (and whereto). In turn, this would prevent any sensible policy for access authorization. There also are purely software-engineering arguments against duplicate private declarations, which generally leads system designers into adopting the idea of controlled export rather than controlled import.

A related restriction is that we assume that a signature can be exported from only one module—the module where it is declared. This includes **public** declarations, which are considered to be exported to all modules. These ideas are formally captured via conditions (i) and (ii) in Definition 14.2, below.

To formulate the notion of coherence, we need a few simple definitions. Let $\mathcal{L}$ be an F-language, **P** be a program with a module structure, and let $\mho$ be a module of **P**. Let $\mathbf{P}_{\mho}$ denote the set of clauses in **P** that belong to $\mho$. The notation $\mathbf{P}^*$ (or $\mathbf{P}^*_{\mho}$) will be used to denote the set of *annotated* ground instances of **P** (resp., $\mathbf{P}_{\mho}$). We shall assume that rule-instantiation preserves annotations of signature expressions that occur in the heads of the rules in **P**, and that instances of the signatures that have different annotations are not merged.[24]

[24]For instance, instantiating X and Y to b in a[**private** : $X \Rightarrow c$; **public** : $Y \Rightarrow c$] yields a[**private** : $b \Rightarrow c$; **public** : $b \Rightarrow c$], where the two identical instances of $b \Rightarrow c$ are kept separately due to the difference in annotations.

Let $\mathbf{I}$ be an H-structure of $\mathbf{P}$. (Note: $\mathbf{I}$ has no annotations, but $\mathbf{P}$ and $\mathbf{P}^*$ have). A ground rule (or query) in $\mathbf{P}^*$ is *active* in $\mathbf{I}$ if all its premises are true in $\mathbf{I}$. If $\mathbf{I}$ is a model, then the heads of all active rules are also true in $\mathbf{I}$.

Definition 14.2 (Coherent Module Structures) A module structure for $\mathbf{P}$ is *coherent* with respect to a canonic H-model, $\mathbf{L}$, if for every data-atom in $\mathbf{L}$ that is covered[25] by a pair of signatures, σ_1 and σ_2, that occur in the heads of some active rules, $r_1, r_2 \in \mathbf{P}^*$ (where $r_1 \equiv r_2$ is possible), the following holds:

(i) both signatures, σ_1 and σ_2, are annotated with the same symbol;[26] and

(ii) the rules r_1 and r_2 belong to the same module.

A module structure is *coherent* if it is coherent with respect to all canonic H-models of $\mathbf{P}$. □

In the above definition, observe that since σ_1 and σ_2 are assumed to cover the same atom, both signatures must refer to the same method (and use the same number of arguments).

Given a program, $\mathbf{P}$, an H-model, $\mathbf{I}$, and a module, $\mho$, of $\mathbf{P}$, the *restriction* of $\mathbf{I}$ to $\mho$, denoted $\mathbf{I}\mid_{\mho}$, is the smallest H-structure that contains the following atoms:

- every data-atom, is-a atom, or a P-atom in $\mathbf{I}$ that occurs in an active rule in $\mathbf{P}^*_{\mho}$; and
- every signature atom in $\mathbf{I}$ that is one of the following:
  - a publicly annotated signature occurring in the head of an active rule in $\mathbf{P}^*$, *i.e.*, in any module;
  - a privately annotated signature that occurs in the head of an active rule in $\mathbf{P}^*_{\mho}$;
  - a signature explicitly exported into $\mho$ from another module, $\mho'$, provided that this signature occurs in the head of an active rule in $\mathbf{P}^*_{\mho'}$.

In other words, $\mathbf{I}\mid_{\mho}$ takes all data atoms that are relevant to the module and all signature atoms that are either derivable within that module or are exported by other modules. For instance, consider a program that includes the three modules (20)–(22) and let $\mathbf{M}$ be its canonic model. The restriction of $\mathbf{M}$ to the module **accounting** contains all data atoms that occur in the active instances of the rules in (22), all signatures declared in **accounting**, and all signatures from modules **faculty** and **person**.

We can now define the notion of type-correctness under meta-annotations as follows:

Definition 14.3 (Type-Correctness with Meta-Annotations) Let $\mathbf{P}$ be an F-program with a coherent module structure. Then $\mathbf{P}$ is said to be *type-correct* (or *well-typed*) if it is well-typed in the usual sense (with respect to Definition 13.5) and if every module of $\mathbf{P}$ is also well-typed.

A module, $\mho$, is *well-typed* if $\mathbf{L}\mid_{\mho}$ is a typed H-model of $\mathbf{P}_{\mho}$ for every canonic model $\mathbf{L}$ of $\mathbf{P}$. □

To illustrate this definition, consider the module structure (20)–(22). This program is well-typed because all signatures used in these modules are properly exported. Suppose now that we ask the following query:

$$?-\ F : faculty[age \rightarrow Y;\ projects \twoheadrightarrow P[account \rightarrow A]] \wedge Y > 50 \qquad (23)$$

[25]Covering is defined in Definitions 13.1 and 13.2.

[26]**export-to(a)** and **export-to(b)** are considered to be the same symbol for this purpose.

This query, when posed in module **person**, violates the encapsulation of the method *account*, since the latter is not exported to **person**. Therefore, we should expect that the modified module **person** is not well-typed. With a caveat, this is, indeed, the case.

To see why, let $\mathbf{L}$ be a canonic model of this program. The restriction $\mathbf{L}\mid_{\mathbf{person}}$ does not contain the signature $project[account \Rightarrow account]$, since it does not appear in the head of any active rule local to **person** and since it is not exported to this module. Thus, if there were even a single faculty, say *fred*, who is, say, 55 years old and who has a project, *bluff*, with an active account, $a123$, the query instance $?-\ fred : faculty[age \rightarrow 55; projects \twoheadrightarrow bluff[account \rightarrow a123]] \wedge 55 > 50$ would be active and, thus, $bluff[account \rightarrow a123]$ would be in $\mathbf{L}\mid_{\mathbf{person}}$. Since this data atom is not covered by any signature in $\mathbf{L}\mid_{\mathbf{person}}$, the latter is not a typed H-structure.

The caveat here is that the notion of type correctness used to model encapsulation has the same drawback as the notion presented in Section 13: being purely semantic, it is weaker than desired. For instance, suppose that no *faculty*-object satisfies the query (23). Then adding the clause (23) to **person** would leave the program well-typed. It will remain well-typed until the first 50+ year old faculty with an active account is inserted. Following this update, the program becomes ill-typed and only then the use of the method *account* inside module **person** would become illegal. It should be emphasized, however, that this problem arises because of the weakness of our notion of type correctness and not due to our treatment of encapsulation.

The above small repertoire of meta-annotations suffices for representing fairly complex encapsulation policies, such as those used in C++. For instance, the declaration **export-to** is akin to the "friend function" declaration in C++. It can also support the "delegation" mechanism found in some object-oriented languages. Other useful encapsulation policies can be represented along the same lines.

15 Inheritance

The concept of inheritance is fundamental in AI and object-oriented programming, and a number of researchers have been working on combining this idea with programming languages. There are two main aspects of inheritance: *structural* inheritance and *behavioral* inheritance. Structural inheritance is a mechanism for propagating method declarations from superclasses to their subclasses. On the other hand, behavioral inheritance propagates what methods actually do rather than how they are declared.

15.1 Structural Inheritance

Structural inheritance is a subject of many works in functional and logic languages. Cardelli [29] considers inheritance in the framework of functional programming. He described a type inference procedure that is sound with respect to the denotational semantics of his system. Sound type inference systems for functional languages were also discussed in [82, 37] and in several other papers.

LOGIN and LIFE [7, 8] incorporate structural inheritance into logic programming via a unification algorithm for ψ-terms, which are complex structures related to signatures in F-logic but that are fundamentally different from the semantical point of view.[27]

[27]There are other important differences as well. For instance, in F-logic, query answers are sets of atoms that are *logically entailed* by the program, while in LIFE query answers are constraints that turn queries into true formulas.

In contrast to the above works, F-logic is a full-fledged logic where structural inheritance is built into the semantics and whose proof theory is *sound* and *complete* with respect to this semantics. Structural inheritance was discussed in Section 7.3.

15.2 Behavioral Inheritance

The main difficulty in dealing with behavioral inheritance is the fact that it is *non-monotonic*, which is mostly due to the property called *overriding* and also because of the phenomenon of *multiple inheritance.*

Overriding means that any explicit definition of a method takes precedence over any definition inherited from a superclass. For instance, consider the following F-program:

$$\begin{array}{ll} royalElephant :: elephant & elephant\,[color \bullet\!\!\to \text{“}grey\text{”}; group \bullet\!\!\to mammal] \\ clyde : royalElephant & royalElephant\,[color \bullet\!\!\to \text{“}white\text{”}] \end{array} \qquad (24)$$

Here *clyde* is a member of the class *royalElephant* and so it inherits $color \bullet\!\!\to white$ from that class (which turns into a non-inheritable data expression, $color \to white$, in the context of *clyde*). In principle, *clyde* could have inherited $color \bullet\!\!\to grey$ from the class *elephant.* However, *clyde* also belongs to a more specific class, *royalElephant*, which has an overriding inheritable property, $color \bullet\!\!\to white$. Likewise, *royalElephant* does not inherit $color \bullet\!\!\to grey$ from *elephant* because of the aforesaid overriding property. On the other hand, *royalElephant* inherits $group \bullet\!\!\to mammal$ as an inheritable property, and *clyde* inherits it as a non-inheritable property, $group \to mammal$.

Note that (24) uses inheritable expressions to indicate that inheritance must take place. We remind from Section 3 that inheritable expressions are used in the following way: when such an expression is inherited by an object, say *clyde*, from a class where *clyde* is a member, it becomes a non-inheritable property of *clyde*, even if *clyde* were to play the role of a class in some other context. In contrast, when an inheritable expression, *e.g.,* $group \bullet\!\!\to mammal$, is inherited by a class, *e.g., royalElephant*, via a subclass relationship, it remains an inheritable expression in that subclass, and so it can be passed down to the subclasses and to the members of that class.

To illustrate the rationale behind the dichotomy of inheritable/non-inheritable data expressions, we shall expand the example in Section 3, where *bob* was represented as a member of the faculty.

Suppose now that information about *bob*'s high school years and his years at Yale needs to be added to the database. One way to accommodate this new information is to create the objects *bobInHighschool* and *bobAtYale* and to make them completely unrelated to the already existing object *bob.* The disadvantage of this approach is that then we will have to duplicate *bob*'s date of birth, gender, etc. A better way is to turn the above new objects into members of the class *bob* as follows:

$$\begin{array}{ll} bobInHighschool\,[graduation \to 1968] & bob : faculty \\ bobAtYale\,[graduation \to 1972] & bobInHighschool : bob \\ faculty\,[highestDegree \bullet\!\!\to phd] & bobAtYale : bob \\ bob\,[birthdate \bullet\!\!\to 1950; gender \bullet\!\!\to \text{“}male\text{”}; address \to \text{“}NewYork\text{”}] & \end{array}$$

Here *bob* is a member of class *faculty*, while *bobInHighschool* and *bobAtYale* are members of class *bob.* Additionally, the second and the third clauses state that *bob* graduated from high school in 1968 and from Yale in 1972. Now, being a member of *faculty*, *bob* inherits $highestDegree \bullet\!\!\to phd$, yielding a new fact, $bob\,[highestDegree \to phd]$. Notice that the property $highestDegree \bullet\!\!\to phd$ inherited from *faculty* turns into a non-inheritable property of *bob*, since inheritable properties are passed down to class members as non-inheritable properties. Because of that, $highestDegree \rightarrow phd$ can no longer be propagated to the members of class *bob*. And, indeed, it makes little sense to attribute a scholarly degree to Bob when he was in high school or a student at Yale. Likewise, *bob*'s property $address \rightarrow$ "$NewYork$" should not be inheritable because, in all likelihood, Bob had a different address while in high school, and certainly a different address while at Yale. On the other hand, *birthdate* and *gender* are specified as inheritable properties, because these characteristics are not likely to change with time.

Turning to multiple inheritance, this phenomenon can be illustrated with the following example that is commonly known as Nixon's Diamond:

$$\begin{array}{ll} nixon : quaker & quaker\,[policy \bullet\!\!\rightarrow pacifist] \\ nixon : republican & republican\,[policy \bullet\!\!\rightarrow hawk] \end{array} \tag{25}$$

As a member of two classes, *quaker* and *republican*, *nixon* can inherit either $policy \bullet\!\!\rightarrow pacifist$ or $policy \bullet\!\!\rightarrow hawk$. However, if one of these properties, say $policy \bullet\!\!\rightarrow pacifist$, is inherited by *nixon* (and becomes a non-inheritable property, $policy \rightarrow pacifist$), then the other property can no longer be inherited because, in the new state, the attribute *policy* is defined on the object *nixon*, and this definition overrides any further inheritance. Thus, in this case, inheritance leads to an indeterminate result: the derivation of $nixon\,[policy \rightarrow pacifist]$ or $nixon\,[policy \rightarrow hawk]$, depending on which inheritance step is done first.

Overriding and multiple inheritance may each cause non-monotonic behavior. A relation of logical entailment, $\models\!\!\approx$, is *non-monotonic* if for some $\mathbf{P}$, ψ, and ϕ, it is possible that $\mathbf{P} \models\!\!\approx \phi$ and $\mathbf{P} \wedge \psi \not\models\!\!\approx \phi$. Classical logic, on the other hand, is monotone, since $\mathbf{P} \models \phi$ always implies $\mathbf{P} \wedge \psi \models \phi$. Non-monotonic logics are generally much more involved, and the corresponding proof theories are rarely complete.

To see why overriding may cause non-monotonic behavior, consider the example in (24), and let $\models\!\!\approx$ be the logical entailment relation that does "the right thing" for inheritance. Let $\mathbf{P}$ denote the program in (24). Previously, we have argued that the following is the intended inference:

$$\mathbf{P} \quad \models\!\!\approx \quad clyde\,[color \rightarrow white] \tag{26}$$

Now, suppose we add $\psi = clyde\,[color \rightarrow silver]$ to our set of assertions. Because of the overriding, the property $color \rightarrow white$ is no longer inferred for *clyde* and, thus, $\mathbf{P} \wedge \psi \;\not\models\!\!\approx\; clyde\,[color \rightarrow white]$.

Multiple inheritance causes non-monotonic behavior for similar reasons. Consider a part of the Nixon's Diamond (25), where it is not yet known that Nixon is a Quaker. In that case, *nixon* would inherit $policy \bullet\!\!\rightarrow hawk$ from the only class, *republican*, where it belongs. However, in a full-blown Nixon's Diamond, one where $nixon : quaker$ is known to hold, $nixon\,[policy \rightarrow hawk]$ is no longer a certainty but, rather, just one of two possibilities.

Nonmonotonic inheritance has been a subject of intensive research (*e.g.*, [41, 100, 101, 50, 25, 99, 65, 45, 64, 88]) for many years. The main difference between our approach and the above works is that we are developing an inheritance theory for a general-purpose object-oriented logical language. In contrast, most of the aforesaid papers tend to study inheritance from a very general, philosophical standpoint, but in isolation from logic programming issues. This is mirrored in the languages they use. First, these languages are mostly propositional and, importantly, do not distinguish between properties (attributes, in our terminology) and classes. Second, these languages are quite limited and, in particular, they are not part of a more general logic programming system. So the important and complex issue of the interaction of inheritance with ordinary logical deduction has not been considered.

The semantics for inheritance to be developed in this section is of the *credulous* variety [50], which means that, in the presence of inheritance conflicts, we are willing to accept multiple canonical models.

This phenomenon can be interpreted in two different ways. One is to view each canonical model as a viable "possible world" and so any one of these models can be chosen non-deterministically; the other interpretation is that only the facts that belong to the intersection of all such models can be trusted. Since our primary concern is programming, we adopt the former view. In other words, when an inheritance conflict occurs due to multiple inheritance, any one of the canonic models can be selected non-deterministically and then used to answer queries.

In a different setting, non-monotonic inheritance was also discussed in [69, 70, 24, 4]. According to these approaches, a logical object (or a class) is a set of rules that represent "knowledge" embodied by the object (resp., class). These rules can be inherited by objects that are placed lower in the hierarchy. Inheritance can be overwritten based on various criteria (*e.g.*, the existence of overriding rules, as in [69, 70], or because of higher-priority rules, as in [24], or due to an exception, as in [4]).

Apart from the very different setup in which behavioral inheritance takes place in these works, there is another fundamental difference with our approach (and, for that matter, with [41, 100] and related works). Namely, in [69, 70, 24], what is inherited is a set of program *clauses*, while in F-logic it is ground data expressions that are passed down the IS-A hierarchy. The latter approach seems to be more flexible. As shown in Section 15.3, F-logic can easily account for inheritance of clauses and for some other forms of inheritance (see Section 15.3) whose representation in the framework of [69, 70, 24] is not obvious to us.

15.2.1 Informal Introduction to the Approach

To integrate inheritance into an object-oriented logic programming system, such as F-logic, the following issues must be addressed:

(i) Interaction between inheritance and ordinary logical deduction;

(ii) Inheritance of set-valued properties;

(iii) Dependence of the IS-A hierarchy on prior inheritance steps; and

(iv) Negative literals in rule bodies.

These issues appear to be closely related and, in fact, it is (ii) and (iii) that makes (i) a hard problem. The difficulty is that after inheritance is done, a program clause may become "active," causing other facts to be derived. This latter derivation may affect the recipient-object of inheritance and, even more curiously, the source-class of inheritance. To illustrate, consider the following program:

$$\begin{array}{l} a : b \\ b\,[attr \bullet\!\!\twoheadrightarrow c] \\ b\,[attr \bullet\!\!\twoheadrightarrow d] \longleftarrow a\,[attr \twoheadrightarrow c] \end{array} \qquad (27)$$

If a inherits $attr \bullet\!\!\twoheadrightarrow c$ from b, the atom $a\,[attr \twoheadrightarrow c]$ is derived and the last rule is activated. This leads to the derivation of $b\,[attr \bullet\!\!\twoheadrightarrow d]$ via the third rule. The question now is whether the inheritance should be "undone" in such a case. Indeed, it can be reasoned that, since $b[attr \bullet\!\!\twoheadrightarrow \{c, d\}]$ holds, the object a should have inherited $attr \bullet\!\!\twoheadrightarrow \{c, d\}$ in its entirety or nothing at all. It seems that different decisions are possible here and, in some cases, the choice may be a matter of taste.

Our choice is to not undo inheritance in this situation and, at the same time, to not inherit the newly derived fact, $b\,[attr \bullet\!\!\twoheadrightarrow d]$. The reason for the former is primarily computational, as this causes less backtracking. The reason for the latter is aesthetic, as it leads to more uniform definitions. Another justification is that once the initial inheritance has taken place, the attribute *attr* is already defined on a. So, by the time $b\,[attr \bullet\!\!\twoheadrightarrow d]$ is derived, inheritance of $attr \bullet\!\!\twoheadrightarrow d$ by a is blocked.

To illustrate complications arising due to (iii), consider the following program:

$$\begin{array}{ll} a : p & p\,[attr \bullet\!\!\rightarrow c] \\ a : t \longleftarrow a\,[attr \rightarrow c] & t\,[attr \bullet\!\!\rightarrow d] \\ t :: p \longleftarrow a\,[attr \rightarrow c] & \end{array} \tag{28}$$

In this example, if a inherits $attr \rightarrow c$ from p the two deductive rules in the program are activated and then both, $a : t$ and $t :: p$, are derived. In other words, t is propelled into the middle of the inheritance path from p to a. What is unusual here is the fact that the attribute *attr* is already defined on t and has a value, d, which is different from c. This means that, if t were sitting on the inheritance path right from the start, the inheritance of $attr \bullet\!\!\rightarrow c$ from p to a would have been blocked, and a would have inherited $attr \rightarrow d$ instead. However, in (28), t was *not* on the inheritance path initially and so the above argument does not apply. Nevertheless, one can argue that the subsequent derivation of $a : t$ and $t :: p$ undermines the basis for the above inheritance step. Thus, the question is, should such inheritance step be undone?

Again, our choice is to not undo such steps. As before, one important reason is computational. The other is semantic simplicity.

The resulting model is called *inheritance-canonic*. The resulting semantics may seem a bit too procedural for one's taste, but, we believe, it captures the intuition rather well. Development of a more declarative semantics is a topic for future work.

In a nutshell, the idea is to decompose each inheritance step into a pair of sub-steps: 1) a "pure" inheritance step, which may introduce new facts but whose output H-structure may no longer be a model of the original program; and 2) a derivation step that turns the result of the previous step into a model. These operations are repeated until inheritance can be done no more.

In this connection, we should mention the recent work [39], which also proposes a semantics for inheritance in a language derived from F-logic. In particular, this work allows inheritance and deduction to interact in certain ways, but it does not deal with negation and dynamically changing IS-A hierarchies. This approach is also fundamentally different from ours in the way semantics is defined. What [39] calls a "canonic model" of a program is actually not a model in the usual sense. Instead, it is a structure where certain "blocked" rules do not have to be satisfied. This aspect of the semantics in [39] is analogous to the work on ordered logic [69, 70], discussed earlier. In contrast, our semantics is more traditional and, in particular, our canonic models are also models in the usual sense.

15.2.2 A Fixpoint Semantics for Non-monotonic Inheritance

In this section, we limit our attention to the case of Horn F-programs. An extension of this semantics to programs with negation in the bodies of rules is given in Appendix B.

Let **I** be an H-structure and $p, m, a_1, \ldots, a_n$ be ground id-terms. We define the result of an application of a method, m, to object p with arguments $a_1, \ldots, a_n$ (for a given invocation type: $\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\rightarrow$, or

$\bullet\!\!\twoheadrightarrow$) as follows:

$$\begin{array}{lcl}
\mathbf{I}(p,\ m@a_1, \ldots, a_n \rightarrow) & = & \{v \mid p[m@a_1, \ldots, a_n \rightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
\mathbf{I}(p,\ m@a_1, \ldots, a_n \bullet\!\!\rightarrow) & = & \{v \mid p[m@a_1, \ldots, a_n \bullet\!\!\rightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
\mathbf{I}(p,\ m@a_1, \ldots, a_n \twoheadrightarrow) & = & \{v \mid p[m@a_1, \ldots, a_n \twoheadrightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
 & & \cup \; \{\, \emptyset \mid \text{if } p[m@a_1, \ldots, a_n \twoheadrightarrow \{\,\}] \in \mathbf{I}\,\} \\
\mathbf{I}(p,\ m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow) & = & \{v \mid p[m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
 & & \cup \; \{\, \emptyset \mid \text{if } p[m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow \{\,\}] \in \mathbf{I}\}
\end{array}$$

Observe that $\mathbf{I}(p,\ m@a_1, \ldots, a_n \twoheadrightarrow)$ is either empty, when $p[m@a_1, \ldots, a_n \twoheadrightarrow \{\,\}]$ is false in $\mathbf{I}$; or it is a set that contains $\emptyset$—a special element that denotes the empty set—and, possibly, some elements from $U(\mathcal{F})$. For scalar invocations, $\mathbf{I}(p,\ m@a_1, \ldots, a_n \rightarrow)$ can either be empty or it can be a subset of the Herbrand universe, $U(\mathcal{F})$ (which does not include $\emptyset$). Note that this set may contain more than one element (even though this is a scalar invocation of m) because of the equality operator: if $t \in \mathbf{I}(p,\ m@a_1, \ldots, a_n \rightarrow)$ and $t \doteq s \in \mathbf{I}$ then also $s \in \mathbf{I}(p,\ m@a_1, \ldots, a_n \rightarrow)$. Similar observations apply to $\mathbf{I}(p,\ m@a_1, \ldots, a_n \bullet\!\!\rightarrow)$ and $\mathbf{I}(p,\ m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow)$.

The use of $\emptyset$ as a special value is a matter of convenience. When $o : cl$ or $o :: cl$ is true and a method, m, returns an empty set on cl, we would like o to inherit $\{\,\}$, unless this inheritance is blocked. In this case, we can say that what is inherited is a special domain value, $\emptyset$, which leads to more uniform definitions.

Definition 15.1 (Inheritance Triggers) Let $\mathbf{I}$ be an H-structure. Consider a pair, $\tau = \langle obj \natural cl,\ m@a_1, \ldots, a_n \bullet\!\!\rightarrow \rangle$, where $obj, cl, m, a_1, \ldots, a_n \in U(\mathcal{F})$ are ground id-terms, and $\natural$ denotes one of the two is-a relations, ":" or "::". We shall say that τ is an *active inheritance trigger* in $\mathbf{I}$ (or just a *trigger*, for brevity) if and only if the following conditions hold:

- $obj \natural cl \in \mathbf{I}$, and there is no intervening class, $mid \in U(\mathcal{F})$, such that $mid \neq cl$, $mid \neq obj$, and $obj \natural mid$, $mid :: cl \in \mathbf{I}$;
- The method m is *defined* in $\mathbf{I}$ as an inheritable property of cl with arguments $a_1, \ldots, a_n$, *i.e.*, $\mathbf{I}(cl,\ m@a_1, \ldots, a_n \bullet\!\!\rightarrow) \neq \{\,\}$; and
- m is *undefined* in $\mathbf{I}$ when invoked on obj with arguments $a_1, \ldots, a_n$. More precisely, if "$\natural$" is ":" (*i.e.*, if obj is a member of cl) then it must be the case that $\mathbf{I}(obj,\ m@a_1, \ldots, a_n \rightarrow) = \{\,\}$. Otherwise, if "$\natural$" is the subclass-relationship "::", then $\mathbf{I}(obj,\ m@a_1, \ldots, a_n \bullet\!\!\rightarrow) = \{\,\}$.

Triggers for set-valued invocations of methods are defined similarly, by replacing $\rightarrow$ with $\twoheadrightarrow$ and $\bullet\!\!\rightarrow$ with $\bullet\!\!\twoheadrightarrow$. □

It is clear from the definition that an inheritance trigger is like a loaded gun poised to fire. Firing a trigger leads to derivation by inheritance of new facts that cannot be derived using classical deduction alone. Once fired, a trigger is "deactivated" and is no longer a trigger in the resulting H-structure.

Given a trigger of the above form, we will say that obj is the *recipient* of the impending inheritance step and cl is the *source*. Note that obj may be a member of class cl or a subclass of cl.

The intention of firing a trigger of the form $\tau = \langle obj : cl,\ m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow \rangle$ is to inherit to the recipient object, obj, the result of the application of m to the source class-object, cl. To capture this idea, we introduce the operator of *firing an active trigger*, τ, in an H-structure, $\mathbf{I}$. This operator is defined as follows:[28]

$$\tau(\mathbf{I}) \quad = \quad \mathbf{I} \cup \{obj\,[m@a_1, \ldots, a_n \twoheadrightarrow v] \mid v \in \mathbf{I}(cl, m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow)\}$$

Note that since obj is a member of cl, the inheritable properties of cl are passed down to obj as non-inheritable properties. On the other hand, when obj is a subclass of cl, *i.e.*, when the trigger has the form $\tau = \langle obj :: cl, m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow \rangle$, then the properties of cl are passed down to obj as inheritable properties:

$$\tau(\mathbf{I}) \quad = \quad \mathbf{I} \cup \{obj\,[m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow v] \mid v \in \mathbf{I}(cl, m@a_1, \ldots, a_n \bullet\!\!\twoheadrightarrow)\}$$

For scalar invocations, this operator is defined similarly, by replacing $\twoheadrightarrow$ with $\rightarrow$ and $\bullet\!\!\twoheadrightarrow$ with $\bullet\!\!\rightarrow$. It is easy to see that τ is a monotonic operator on H-structures in which τ is an active trigger.

When a trigger is fired, previously inactive program rules may become applicable and more facts may be further derived via ordinary classical deduction. For instance, consider the following program $\mathbf{P}$:

$$\begin{array}{l} a : b \\ b\,[attr \bullet\!\!\twoheadrightarrow c] \\ a\,[attr \twoheadrightarrow d] \longleftarrow a\,[attr \twoheadrightarrow c] \end{array}$$

The minimal model here consists of $a : b$ and $b\,[attr \twoheadrightarrow c]$. Firing the trigger $\langle a : b, attr \bullet\!\!\twoheadrightarrow \rangle$ adds a new fact, $a\,[attr \twoheadrightarrow c]$, and the resulting H-structure is no longer a model, as it violates the last rule. Therefore, to obtain a model that accommodates the inherited fact, we have to apply the last rule, which derives $a\,[attr \twoheadrightarrow d]$. Note that derivations performed after firing a trigger may affect both the recipient object of the trigger (as shown above) and the source-class (as shown earlier, in (27)).

These ideas are captured via the notion of *one-step inheritance.*

Definition 15.2 (One Step Inheritance Transformation) Let $\mathbf{P}$ be a Horn F-program, $\mathbf{I}$ and $\mathbf{J}$ be H-models of $\mathbf{P}$, and let τ be a trigger in $\mathbf{I}$. We say that $\mathbf{J}$ is obtained from $\mathbf{I}$ via τ by *one step of inheritance*, written $\Im^{\tau}_{\mathbf{P}}(\mathbf{I}) = \mathbf{J}$, if $\mathbf{J}$ is an H-structure that is minimal among the H-models of $\mathbf{P}$ that contain $\tau(\mathbf{I})$. (It is easy to see that, since $\mathbf{P}$ is Horn, $\mathbf{J}$ is the intersection of all H-models of $\mathbf{P}$ that contain $\tau(\mathbf{I})$ and, thus, it is unique.) $\square$

Note that $\Im^{\tau}_{\mathbf{P}}$ is monotonic with respect to H-structures that have τ as a trigger, since τ is also monotonic on such structures.

We are now ready for the notion of canonical models in the presence of inheritance. At the heart of this notion is what we call *The Principle of Minimally Necessary Inheritance*, which refers to the preference of ordinary logical deduction over deduction by inheritance. This means that inheritance ought to be attempted only when ordinary deduction is no longer possible.

The principle of minimally necessary inheritance is embodied in the one-step inheritance operator. Indeed, if $\mathbf{M}$ is a model of $\mathbf{P}$ then $\Im^{\tau}_{\mathbf{P}}(\mathbf{M})$ is a model that extends $\mathbf{M}$ with minimally possible inheritance. Therefore, we define inheritance-canonical models to be models that are obtained from the minimal model via a sequence of such extensions. In general, this sequence may be transfinite.

[28]Note that $\mathbf{I}(cl, m@a_1, \ldots, a_n \twoheadrightarrow)$ may contain the special element, $\emptyset$. So, when $v = \emptyset$, $obj\,[m@a_1, \ldots, a_n \twoheadrightarrow v]$ should be read as $obj\,[m@a_1, \ldots, a_n \twoheadrightarrow \{\,\}]$.

Definition 15.3 (Canonical Models with Inheritance) Let $\mathbf{I}$ be an H-structure. We shall use $\Im^{*}_{\mathbf{P}}(\mathbf{I})$ to denote any H-structure, $\mathbf{M}$, for which there is a (possibly transfinite) sequence of models, $\mathbf{M}_0 \subset \mathbf{M}_1 \subset \ldots \subset \mathbf{M}_\alpha$, where α is an ordinal number, such that:

(i) $\mathbf{M}_0$ is $\mathbf{I}$ and $\mathbf{M}_\alpha$ is $\mathbf{M}$;

(ii) $\mathbf{M}_\alpha$ has no active triggers;

(iii) if γ is a non-limit ordinal, then $\mathbf{M}_\gamma = \Im^{\tau}_{\mathbf{P}}(\mathbf{M}_{\gamma-1})$ for some trigger τ in $\mathbf{M}_{\gamma-1}$; and

(iv) if γ is a limit ordinal, then $\mathbf{M}_\gamma = \cup_{\beta<\gamma}\mathbf{M}_\beta$.

Let $\mathbf{P}$ be a Horn F-program. An H-structure, $\mathbf{M}$, of $\mathbf{P}$ is *inheritance-canonical* (or *ι-canonical*) if it can be represented as $\Im^{*}_{\mathbf{P}}(\mathbf{L})$, where $\mathbf{L}$ is the minimal model of $\mathbf{P}$. □

We shall see from the next lemma that ι-canonical structures are, in fact, models of $\mathbf{P}$. Furthermore, the operator $\Im^{*}_{\mathbf{P}}$ defined above is non-deterministic in the sense that different outcomes may result from different choices of triggers at Step (iii). Therefore, a program may have many ι-canonic model, which is different from the classical case where every Horn program has exactly one canonic model (which coincides with its minimal model).

Lemma 15.4 *Let* $\mathbf{P}$ *be a Horn F-program and let* $\mathbf{I}$ *be an H-structure for* $\mathbf{P}$. *Let, further,* τ *be an active trigger in* $\mathbf{I}$. *Then* $\Im^{\tau}_{\mathbf{P}}(\mathbf{I}) = (T_{\mathbf{P}} \uparrow \omega)(\tau(\mathbf{I}))$, *where* $T_{\mathbf{P}} \uparrow \omega$ *denotes the countably-infinite iteration of* $T_{\mathbf{P}}$. *Moreover,* $\Im^{\tau}_{\mathbf{P}}(\mathbf{I})$ *and every one of the* $\Im^{*}_{\mathbf{P}}(\mathbf{I})$*'s is a model of* $\mathbf{P}$.

Proof: The first two claims are direct consequences of the definitions. The last claim follows from the fact that each $\mathbf{M}_\gamma$ in Definition 15.3 is a model (since so is every $\Im^{\tau}_{\mathbf{P}}(\mathbf{I})$), and $\Im^{*}_{\mathbf{P}}(\mathbf{I})$ is a union of a growing chain of models of a universal set of clauses. □

In general, reaching a canonical model in Definition 15.3 may take a transfinite number of iterations, because each application of $\Im^{\tau}_{\mathbf{P}}$ can introduce countably many new triggers and, for some trigger orderings, the number of iterations may reach any countable ordinal. However, for "fair" orderings, the number of iterations will be no more than ω.

Lemma 15.5 (Fair Trigger Ordering) *There are trigger orderings for which the iterative process of Definition 15.3 will compute the ι-canonic model after ω iterations.*

Proof: One fair trigger ordering can be defined like this. Every time a new iteration is performed with a $\Im^{\tau}_{\mathbf{P}}$ operator, each newly introduced trigger is assigned a pair of natural numbers, (n, m), where n is the iteration number and m is a unique index (within the set of the new triggers, which is countable).

We can order the set of pairs $\{(n, m) \mid n, m > 0\}$ using Kantor's diagonalization process and then apply triggers in that order.[29] Let $\mathbf{M}_{n,m}$ denote the model that is obtained in (iii) of Definition 15.3 by applying $\Im^{\tau}_{\mathbf{P}}$ with τ labeled with the pair (n, m).

It remains to show that $\mathbf{M} = \cup_{n,m>0}\mathbf{M}_{n,m}$ is a model of $\mathbf{P}$ with no active triggers. First, clearly, $\mathbf{M}$ is a model of $\mathbf{P}$. To show that it has no triggers, suppose, to the contrary, that τ is one. Then it

[29]Some triggers may become deactivated on the way because of rule applications and other trigger firings. In such a case, the trigger is simply skipped over.

must have become active in some $\mathbf{M}_{k,l}$. By construction, τ must have been assigned some pair, (s,t), and, therefore, it must have been fired in $\mathbf{M}_{s,t}$. $\Box$

We shall now illustrate the notions of one-step inheritance and of ι-canonical model via a series of examples. The following two examples are derived from the Nixon's Diamond example.

Consider the following program, $\mathbf{P}$:

$$nixon : republican \qquad\qquad republican\,[policy \bullet\!\!\rightarrow hawk]$$

This states that *nixon* belongs to the class *republican* and that, generally, republicans are hawks. The "interesting" H-models of $\mathbf{P}$ are summarized below:

object	**L**	**M**
republican	$policy \bullet\!\!\rightarrow hawk$	$policy \bullet\!\!\rightarrow hawk$
nixon	—	$policy \rightarrow hawk$
is-a	$nixon : republican$	$nixon : republican$

Here $\mathbf{L}$ is a minimal model of $\mathbf{P}$, but it is not ι-canonical. To see this, note that $\mathbf{M} = \tau(\mathbf{L}) = \Im^{\tau}_{\mathbf{P}}(\mathbf{L})$, where $\tau = \langle nixon : republican, policy \bullet\!\!\rightarrow \rangle$ is a trigger, *i.e.*, $\mathbf{M}$ is derived from $\mathbf{L}$ by one inheritance step. It is easy to verify that $\mathbf{M}$ is one (and the only) ι-canonical H-model of the program. If, however, $\mathbf{P}$ contained the fact $C = nixon\,[policy \rightarrow pacifist]$, then no inheritance would have taken place, because the more specific value, *pacifist*, overrides the inheritance of the value *hawk* from the class *republican* (which means that τ is not a trigger in the minimal model of $\mathbf{P} \cup \{C\}$).

The next example illustrates multiple inheritance. Let the program $\mathbf{P}$ be:

$$\begin{array}{ll} nixon : republican & \qquad republican\,[policy \bullet\!\!\rightarrow hawk] \\ nixon : quaker & \qquad quaker\,[policy \bullet\!\!\rightarrow pacifist] \end{array}$$

The following H-models are of interest here:

object	**L**	$\mathbf{M}_1$	$\mathbf{M}_2$
republican	$policy \bullet\!\!\rightarrow hawk$	$policy \bullet\!\!\rightarrow hawk$	$policy \bullet\!\!\rightarrow hawk$
quaker	$policy \bullet\!\!\rightarrow pacifist$	$policy \bullet\!\!\rightarrow pacifist$	$policy \bullet\!\!\rightarrow pacifist$
nixon	—	$policy \rightarrow hawk$	$policy \rightarrow pacifist$
is-a	$nixon : republican$	$nixon : republican$	$nixon : republican$
	$nixon : quaker$	$nixon : quaker$	$nixon : quaker$

Here $\Im^{\tau}_{\mathbf{P}}(\mathbf{L}) = \tau(\mathbf{L}) = \mathbf{M}_1$ and $\Im^{\kappa}_{\mathbf{P}}(\mathbf{L}) = \kappa(\mathbf{L}) = \mathbf{M}_2$, where $\tau = \langle nixon : republican, policy \bullet\!\!\rightarrow \rangle$ and $\kappa = \langle nixon : quaker, policy \bullet\!\!\rightarrow \rangle$. Furthermore, it is easy to check that $\mathbf{M}_1$ and $\mathbf{M}_2$ are both ι-canonical models of $\mathbf{P}$, while the model $\mathbf{L}$ is minimal but not ι-canonical.

We thus see that both, $\mathbf{M}_1$ and $\mathbf{M}_2$, are obtained from $\mathbf{L}$, albeit by firing different triggers. These triggers are associated with the same attribute, *policy*, and the same recipient object, *nixon*, but with different source classes, *quaker* and *republican*. For this reason, inheritance steps obtained by firing each trigger separately cannot be combined, because firing of one trigger deactivates the other, and vice versa. The existence of a pair of triggers with these properties is a formal explanation for the phenomenon of multiple inheritance.

The next F-program, $\mathbf{P}$, illustrates one of the many ways in which inheritance can interact with deduction:

$$\begin{array}{ll} p : q & q\,[attr \mathrel{\bullet\!\!\twoheadrightarrow} a] \\ r : q & r\,[attr \twoheadrightarrow b] \longleftarrow p\,[attr \twoheadrightarrow a] \end{array} \tag{29}$$

There are two triggers, $\tau = \langle p : q,\ attr \mathrel{\bullet\!\!\twoheadrightarrow} \rangle$, and $\kappa = \langle r : q,\ attr \mathrel{\bullet\!\!\twoheadrightarrow} \rangle$. Observe that applying τ will activate the last rule. Let $\mathbf{L}$ be the minimum model of $\mathbf{P}$, and let $\mathbf{M}_1, \mathbf{M}_2$ be the two models that emerge after firing one of the above two triggers, *i.e.*, $\Im_{\mathbf{P}}^{\tau}(\mathbf{L}) = \mathbf{M}_1$ and $\Im_{\mathbf{P}}^{\kappa}(\mathbf{L}) = \mathbf{M}_2$. It is easy to verify, that applying τ deactivates κ, but firing κ does not deactivate τ. Therefore, there is no trigger to fire in $\mathbf{M}_1$ and no further inheritance is possible. However, τ can still be fired in $\mathbf{M}_2$, leading to another model, $\mathbf{M}_3 = \Im_{\mathbf{P}}^{\tau}(\mathbf{M}_2)$. The following table shows the relevant parts of these models:

	$\mathbf{L}$	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$
object				
q	$attr \mathrel{\bullet\!\!\twoheadrightarrow} a$	$attr \mathrel{\bullet\!\!\twoheadrightarrow} a$	$attr \mathrel{\bullet\!\!\twoheadrightarrow} a$	$attr \mathrel{\bullet\!\!\twoheadrightarrow} a$
p	—	$attr \twoheadrightarrow a$	—	$attr \twoheadrightarrow a$
r	—	—	$attr \twoheadrightarrow a$	$attr \twoheadrightarrow a$
r	—	$attr \twoheadrightarrow b$	—	$attr \twoheadrightarrow b$
is-a	$p : q,\ r : q$	$p : q,\ r : q$	$p : q,\ r : q$	$p : q,\ r : q$

Both $\mathbf{M}_1$ and $\mathbf{M}_3$ are ι-canonical models.

The existence of two ι-canonic models for the above program may be somewhat discomforting, because a case can be made that trigger τ should have been fired before κ. Indeed, firing the former deactivates the latter, yielding $\mathbf{M}_1$, a subset of $\mathbf{M}_3$. However, in view of the inherent nondeterminism in our semantics, it is unclear why $\mathbf{M}_3$ would be a less legitimate model than $\mathbf{M}_1$.

The following example illustrates the phenomenon of inheritance in dynamically changing IS-A hierarchies. Let the program, $\mathbf{P}$, be:

$$\begin{array}{ll} a : p & p\,[attr_1 \mathrel{\bullet\!\!\twoheadrightarrow} c] \\ a : t \longleftarrow a\,[attr_1 \twoheadrightarrow c] & t\,[attr_1 \mathrel{\bullet\!\!\twoheadrightarrow} d; attr_2 \mathrel{\bullet\!\!\twoheadrightarrow} e] \\ t :: p \longleftarrow a\,[attr_1 \twoheadrightarrow c] & \end{array}$$

Let $\mathbf{L}$ be the minimum model of $\mathbf{P}$. Then $\tau = \langle a : p,\ attr_1 \mathrel{\bullet\!\!\twoheadrightarrow} \rangle$ is the only active trigger in $\mathbf{L}$. Firing τ causes $a : t$ and $t :: p$ to be derived. This, in turn, activates the trigger $\kappa = \langle a : t,\ attr_2 \mathrel{\bullet\!\!\twoheadrightarrow} \rangle$ and $a\,[attr_2 \twoheadrightarrow e]$ is derived by inheritance. Note that there is no inheritance of $attr_1 \mathrel{\bullet\!\!\twoheadrightarrow} d$ from t to a, since $attr_1$ is already defined on a.

The most interesting aspect of this example is that the very act of inheritance alters the IS-A hierarchy on which inheritance so heavily depends. Moreover, the derivation of $a : t$ and $t :: p$ propels the fact $t\,[attr_1 \mathrel{\bullet\!\!\twoheadrightarrow} d]$ with an inheritable property right in the middle of the path through which a had inherited $attr_1 \mathrel{\bullet\!\!\twoheadrightarrow} c$ from p. This issue was already discussed in connection with Example (28) earlier, where we argued that the above inheritance step should not be undone, despite the changes in the IS-A hierarchy.

To see how this dynamic situation is taken care of in our semantics, consider the following table that presents the relevant parts of the models of interest:

object	$\mathbf{L}$	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$
p	$attr_1 \bullet\!\!\twoheadrightarrow c$	$attr_1 \bullet\!\!\twoheadrightarrow c$	$attr_1 \bullet\!\!\twoheadrightarrow c$	$attr_1 \bullet\!\!\twoheadrightarrow c$
t	$attr_1 \bullet\!\!\twoheadrightarrow d$	$attr_1 \bullet\!\!\twoheadrightarrow d$	$attr_1 \bullet\!\!\twoheadrightarrow d$	$attr_1 \bullet\!\!\twoheadrightarrow d$
t	$attr_2 \bullet\!\!\twoheadrightarrow e$	$attr_2 \bullet\!\!\twoheadrightarrow e$	$attr_2 \bullet\!\!\twoheadrightarrow e$	$attr_2 \bullet\!\!\twoheadrightarrow e$
a	—	$attr_1 \twoheadrightarrow c$	$attr_1 \twoheadrightarrow d$	$attr_1 \twoheadrightarrow c$
a	—	$attr_2 \twoheadrightarrow e$	$attr_2 \twoheadrightarrow e$	$attr_2 \twoheadrightarrow e$
a	—	—	—	$attr_1 \twoheadrightarrow d$
is-a	$a : p$	$a : t,\ t :: p$	$a : t,\ t :: p$	$a : t,\ t :: p$

Among the above models, only $\mathbf{M}_1$ is ι-canonical; it is obtained from $\mathbf{L}$ by firing the previously defined triggers τ and κ.

The subtle point here is that if $a : t$ and $t :: p$ were true in the model $\mathbf{L}$ in the beginning, the atom $t\,[attr_1 \bullet\!\!\twoheadrightarrow d]$ lurking on the path $a : t :: p$ would have blocked the inheritance of $attr_1 \bullet\!\!\twoheadrightarrow c$ from p to a. However, in our semantics, a *post factum* blocking of an inheritance path, such as above, does not invalidate earlier inheritance steps, even if these steps had been using the same inheritance path before.

15.3 Strategies for Overriding Behavioral Inheritance

In this section, we illustrate various ways in which F-logic inheritance can model overriding strategies used in other object-oriented programming languages.

Pointwise Overriding

Suppose that the following rules provide a definition for the method *grade* in class *instructor*:

$$\begin{array}{rcl} instructor\,[grade \,@\, Stud, vlsi \bullet\!\!\rightarrow G] & \longleftarrow & \cdots \\ instructor\,[grade \,@\, Stud, db \bullet\!\!\rightarrow G] & \longleftarrow & \cdots \\ \ddots & & \end{array}$$

This is a "default" definition of a method, *grades*, which is supposed to provide a way of computing students' grades in different courses. An instructor, say *bob* (assuming *bob* : *instructor* holds), can access this method via a query of the form:

$$?-\ bob\,[grade \,@\, mary, vlsi \rightarrow G]$$

Suppose $instructor\,[grade \,@\, mary, vlsi \bullet\!\!\rightarrow 90]$ holds in class *instructor*. Then $grade \,@\, mary, vlsi \bullet\!\!\rightarrow 90$ will be inherited by *bob*, yielding the atom $bob\,[grade \,@\, mary, vlsi \rightarrow 90]$. Similar inheritance will take place for *db* and other courses.

Suppose now that instructors are allowed to modify the default grade computation by defining their own algorithms. Thus, for instance, *bob* may define his own policy in the database course:

$$bob\,[grade \,@\, Stud, db \rightarrow G]\ \longleftarrow\ \cdots$$

This would provide an explicit definition for *grade* in the object *bob* when this method is invoked with arguments *stud* and *db*, where *stud* represents an arbitrary student. Thus, this explicit definition will override the default. However, notice that overriding has taken place only for some *specific* arguments, while the default definition is still being used for other arguments (*e.g.*, when the course is *vlsi*).

This property of F-logic inheritance is called *pointwise* overriding. As we have seen, pointwise overriding provides a great deal of flexibility by letting the user modify methods over subdomains. This should be contrasted with most other languages where methods are either overwritten in their entirety or not overwritten at all. This is called *global* overriding and is discussed next.

Global Method Overriding

In addition to pointwise overriding, it is easy to arrange for methods to be overwritten globally, as in most other object-oriented languages. Consider the following example:

$$\begin{array}{l} bob : instructor \\ instructor\,[gradingMethod \bullet\!\!\to defaultGrading] \\ X\,[defaultGrading\,@\,Stud, Crs \to G] \longleftarrow X : instructor \wedge \cdots \\ bob\,[gradingMethod \to bobGrading] \\ bob\,[bobGrading@Stud, Crs \to G] \longleftarrow \cdots \\ X\,[grade\,@\,Stud, Crs \to G] \longleftarrow X : instructor\,[gradingMethod \to M;\ M\,@\,Stud, Crs \to G] \end{array} \qquad (30)$$

In this example, the value of *gradingMethod* is the name of a method to be used for grade computation. This method is defined (in the third clause) for each *instructor*-object. The fourth clause specifies the name of the grading method that *bob* prefers to use. This method is defined on the object *bob* using the fifth clause.

The last clause, then, defines *grade* using the method M obtained from the *gradingMethod* attribute. Normally, this value would be inherited from the class *instructor* and will be *defaultGrading*. However, for *bob*, the default value of *gradingMethod* is overwritten. Therefore, when X is bound to *bob*, M will be bound to *bobGrading* and so *grade* will behave exactly like *bobGrading*. In contrast, if *mary* is also an instructor, but she did not override the attribute *gradingMethod*, the value for M in the last clause will be inherited from *instructor* and will thus be *defaultGrading*. Therefore, when X is bound to *mary*, the method *grade* will behave exactly like *defaultGrading*.

This shows how global method overriding used in most object-oriented systems can be modeled via F-logic's pointwise overriding.

User-Controlled Inheritance

The semantics of inheritance described in this section is non-deterministic, *i.e.*, in case of an inheritance conflict, the system will fire one of the active triggers non-deterministically. In some cases, however, the user may want to have more control over trigger-firing. This approach is common in many programming languages, such as Eiffel [78] or C++ [97], where the programmer has to resolve inheritance conflicts explicitly.

User-defined inheritance can be expressed in F-logic as follows. To resolve a conflict, say, in a binary method, *mthd*, we can first parameterize *mthd* by defining a family of methods, *mthd*($Class$):

$$Class\,[mthd(Class)\,@\,X, Y \bullet\!\!\to Z] \longleftarrow Class\,[mthd\,@\,X, Y \bullet\!\!\to Z]$$

Now, let $sup_1, \ldots, sup_n$ denote all the classes where obj belongs and to which the method $mthd$ applies. Then obj will inherit the methods $mthd(sup_1), \ldots, mthd(sup_n)$. This time, however, there is no conflict between these methods, due to parameterization. So, obj can invoke $mthd$ at any superclass of obj, *e.g.*,

$$?- \quad obj\,[mthd(sup_7)\,@\,a, b \rightarrow Z]$$

This is analogous to the use of scope resolution operators in object-oriented languages, such as C++.

16 Further Issues in Data Modeling

While F-logic has a wide range of object-oriented concepts represented directly in its semantics, several other ideas have been left out. These include complex values, path expressions, and others. In this section, we propose ways to model these missing concepts using the rich arsenal of the built-in features available in F-logic.

16.1 Existing and Non-existent Objects

It is easy to verify that $\mathbf{P} \models t[\,]$ holds for any id-term, t, and any program, $\mathbf{P}$. In this sense, in F-logic, any object exists as long as it can be named. On the other hand, in databases and other object-oriented systems, it is a common practice to distinguish between existing objects and those that do not exist. To come into existence, even an empty object must first be created, which seems to be in stark contrast to F-logic where trivial atoms, such as $t[\,]$ above, can be used at any time.

Fortunately, the concept of "existence" can be easily expressed as a property of an object, using a designated attribute, say *exists*. For instance, if $t\,[exists \rightarrow true]$ is in the canonical model of $\mathbf{P}$ then the object is said to exist. Otherwise, it does not.

A preprocessor can modify every clause in $\mathbf{P}$ in such a way that every object molecule that contains a data expression will be augmented with another data expression, $exists \rightarrow true$. Queries should be modified accordingly. For instance, the query $?-\ X$ will be changed to $?-\ X\,[exists \rightarrow true]$, which ensures that only "existing" objects will be retrieved.

The very action of object creation, *i.e.*, adding the property $exists \rightarrow true$, involves *Transaction Logic*. This logic is briefly illustrated in Section 17.4; see [21] for the full treatment.

16.2 Empty Sets vs. Undefined Values

The semantics of set-valued methods distinguishes between cases when the value of a method is an empty set and cases when the method is undefined. Although sometimes the user may not care about this distinction, the difference is important. For instance, asserting $john\,[children \twoheadrightarrow \{\,\}]$ means that *john* has no children (provided that there are no other statements of the form $john\,[children \twoheadrightarrow \cdots]$). In contrast, when $john\,[children \twoheadrightarrow \{\,\}]$ does not hold, this means that it is *unknown* whether *john* has children. As explained earlier, method undefinedness is analogous to null values in classical database theory.

The problem is that when the user poses a query, such as

$$?-\ john\,[children \twoheadrightarrow X] \tag{31}$$

and X gets no bindings, it is unclear whether this is because John has no children or because this attribute has a null value. To verify this, the user will have to ask one more query: $?-\ john\,[children \twoheadrightarrow \{\,\}]$.

Clearly, asking two queries to find out one simple thing is a questionable practice, and it would be highly desirable if a single query (31) were sufficient. One easy way to fix the problem is to introduce a special constant, $\emptyset$, into every F-language, along with the following axiom schemata:

$$X\,[\,M\,@\,A_1,\ldots,A_n \twoheadrightarrow \emptyset\,] \longleftarrow X[\,M@A_1,\ldots,A_n \twoheadrightarrow \{\,\}\,]$$
$$X\,[\,M\,@\,A_1,\ldots,A_n \bullet\!\!\twoheadrightarrow \emptyset\,] \longleftarrow X[\,M@A_1,\ldots,A_n \bullet\!\!\twoheadrightarrow \{\,\}\,]$$

where $n = 0, 1, 2, \ldots$. Technically speaking, these axioms ensure that methods never return empty sets and that the empty set is represented by the singleton set that consists of $\emptyset$ alone.

16.3 Complex Values

Several recent works [16, 3, 71] have suggested that pure object-based languages may be burdensome when identity of certain objects is immaterial for the program. We, too, made this point in Section 6, advocating the use of predicates on a par with objects (recall that predicates can be viewed as objects with "uninteresting" id's). However, it may sometimes be convenient to go beyond simple predicates and allow object-like structures that have no associated object id's. Such structures are called *complex values* [16, 3, 71]. For instance, adapting an example from [71], we could write:

$$eiffelTower\,[name \to \text{“}Eiffel\ Tower\text{”};\ address \to [city \to paris;\ street \to champDeMars]\,]$$

Here $[city \to paris;\ street \to champDeMars]$ is a "complex value" representing an address. In O_2 [71], such constructs have no oid and thus are distinct from objects. The rationale is that the user does not have to shoulder the responsibility for assigning id's to objects that are not going to be accessed through these id's anyway.

In the spirit of [3, 71], we may try to adapt the concept of complex values to F-logic as follows:

(i) Any id-term is a *value*;

(ii) $\{Val_1, \ldots, Val_n\}$ is a value, provided that each Val_i is a value;

(iii) $[\ldots,\ Attr \rightsquigarrow Val,\ \ldots]$ is a value, where *Attr* is an id-term,
Val is a value, and $\rightsquigarrow$ is one of the four arrows, $\to$, $\bullet\!\!\to$, $\twoheadrightarrow$, or $\bullet\!\!\twoheadrightarrow$,
that may occur in data expressions.

To account for complex values, the definition of data expressions in Section 4 should be modified to allow complex values (rather than just oid's) to appear on the right-hand side of data expressions.

First, note that Item (i) above is already legal in data expressions. Also, (ii) is "almost" legal: it is a generalization of the set-construct, where previously only sets of id-terms were allowed on the right-hand side in a data expression.

The next question is the semantics of values introduced in this way. The answer depends on the intended usage of values. If they are taken too literally, as terms assignable to logical variables, this could jeopardize the proof theory of our logic. Indeed, by (i), variables are values and so they can unify with other values. But then an atom such as $obj\,[attr \to X]$ should be unifiable with, say,

$$obj\,[attr \twoheadrightarrow \{a, b, c\}\,]$$

where X would unify with the set $\{a, b, c\}$. In other words, (i) entails that variables can range over sets, thereby rendering the semantics second-order in the sense of [34].

Fortunately, the experience with object-oriented database systems, such as Orion [61] and O_2 [71], suggests that complex values are used in fairly restricted ways—primarily as *weak entities* of the Entity-Relationship Approach. In other words, they are used as "second-rate objects," accessible only through other objects.

This leads to the conclusion that values are used not for their own sake, but rather as a matter of convenience, to let the user omit oid's when they are immaterial. To some extent, this is similar to Prolog's don't-care variables represented by underscores. By analogy, we could permit the use of don't-care object id's. For instance, we can re-cast the previous example as follows:

$$eiffelTower\,[name \rightarrow \text{“}\textit{Eiffel Tower}\text{”};\; address \rightarrow \_[city \rightarrow paris;\; street \rightarrow champDeMars]\,]$$

It remains to decide on a suitable semantics of the don't-care symbol, "_". Let

$$T\,[\,ScalM\,@\,S_1, \ldots, S_n \rightarrow \_[\ldots]\,]$$

be a molecule with a don't-care symbol. It's meaning can be given through the following encoding:

$$T\,[\,ScalM\,@\,S_1, \ldots, S_n \rightarrow val_n(T, ScalM, S_1, \ldots, S_n)[\ldots]\,]$$

where val_n is a new $(n+2)$-ary function symbol, one that is specifically chosen for this kind of encoding. We do not provide interpretation for don't-care symbols inside the set-construct, as in

$$T\,[\,SetM\,@\,R_1, \ldots, R_m \twoheadrightarrow \{\ldots, \_[\ldots], \ldots\}\,]$$

because the meaning and the utility of "_" in this context is not clear. However, below, we introduce another don't-care symbol, "$*$", which can occur inside sets.

The need for don't-care oid's was also emphasized in ILOG [51], where a special "$*$"-notation was introduced for this purpose. An *invention atom*, $p(*, t_1, \ldots, t_n)$, of ILOG is semantically close to an F-molecule of the form

$$p\text{-}tuple(t_1, \ldots, t_n) : p\,[arg_1 \rightarrow t_1;\; \ldots;\; arg_n \rightarrow t_n]$$

where, as in Section 6, *p-tuple* is a function symbol specifically chosen to represent "invented objects" of class p. In other words, invention atoms of [51] are essentially *value-based* P-molecules of F-logic.[30]

The "$*$"-style don't-care symbols can be useful for modeling value-based objects, such as relations with mnemonically meaningful attribute names. For instance, we could represent the SUPPLIER relation via a class, *suppl*, declared as follows:

$$suppl\,[name \Rightarrow string;\; parts \Rightarrow\!\!\!\!\!\Rightarrow part;\; addr \Rightarrow string]$$

Then we could define the contents of this relation as follows:

$$* : suppl\,[name \rightarrow \text{“XYZ Assoc.”};\; parts \twoheadrightarrow all\text{-}in\text{-}one;\; addr \rightarrow \text{“Main St., USA”}\,]$$
$$* : suppl\,[name \rightarrow \text{“Info Ltd.”};\; parts \twoheadrightarrow know\text{-}how;\; addr \rightarrow \text{“P.O. Box OO”}\,]$$
$$\ddots$$

[30]However, ILOG is not an entirely value-based language, since there is a way (albeit indirect) to create multiple copies of distinct objects with the same contents.

Here, the symbol "$*$" is used as a short-hand notation intended to relieve programmers from the tedium of specifying oid's explicitly (which are *suppl-tuple*("*XYZ Assoc.*", "*Main St. USA*") and *suppl-tuple*("*Info Ltd.*", "*P.O. Box OO*") in our case).

It should be clear that the semantics of "$*$"-style objects is different from the semantics of "_"-style objects. For instance, "$*$"-style objects are entirely *value-based*, as their id's are uniquely determined by their structure. Therefore, a "$*$"-style object can live its own, independent life. Its oid is replaced by "$*$" simply because this oid is uniquely determined by the values of the scalar attributes of the object, making the explicit mention of the id redundant. In contrast, "_"-objects are *not* value-based. The identity of such an object depends mostly on the syntactic position occupied by this object inside another molecule. What the two don't-care styles share, though, is that in both cases the explicit mention of the object identity is redundant, as it can be determined from the context or from the object's structure.

16.4 Path Expressions

Many calculus-like object-oriented languages (*e.g.*, [112, 15]) use syntactic constructs known as *path expressions*. For instance, to select employees working in "*CS*" departments, one could use a path expression $X.affiliation.dname \doteq$ "CS" instead of $X\,[affiliation \rightarrow D\,[dname \rightarrow \text{“}CS\text{”}]\,]$.

The idea of path expressions was significantly extended in [54], where it was used to build an object-oriented extension of SQL. In the syntax of [54], the expression (ix) of Figure 4 can be written thus: $X.affiliation.dname\,[\text{“}CS\text{”}]$. Here, "$CS$" plays the role of a *selector* that selects sequences of objects x, y, z such that $x[affiliation \rightarrow y]$, $y[dname \rightarrow z]$, and $z =$ "CS" are true.

Although path expressions are not explicitly part of F-logic syntax, they can be added as a syntactic sugar to help reduce the number of variables in F-programs (and also because this notation is more familiar to database users). For instance, we could write:

$$?-\ X : empl \wedge X.name\,[N] \wedge X.friends.name\,[\text{“}Bill\text{”}] \wedge X.affiliation.dname\,[\text{“}CS\text{”}]$$

instead of the bulkier query:

$$?-\ X : empl\,[name \rightarrow N;\ friends \twoheadrightarrow F\,[name \rightarrow \text{“}Bill\text{”}];\ affiliation \rightarrow D\,[dname \rightarrow \text{“}CS\text{”}]\,]$$

Formal translation of path expressions (as defined in [54]) into F-logic is left to the reader as an easy exercise; the above example already illustrates the main ideas behind such translation. A more complete discussion of path expressions in F-logic can be found in [43].

16.5 Primary Classes and Immediate Superclasses

Many object-oriented systems have the notion of object's *primary class*, which is the unique, lowest class in the class hierarchy where the object belongs. Most systems also assume discrete class hierarchies—those where every class, c, has a set $super_c$ of *immediate* superclasses with the following properties: (1) every strict superclass of c is also a superclass of some class in $super_c$; and (2) distinct objects in $super_c$ are incomparable in the class hierarchy.

The assumption about primary classes and discrete class hierarchies simplifies inheritance and type checking, and it seems adequate for much of data and knowledge modeling.

To represent these relationships in F-logic, we can introduce a pair of atoms, $a :_1 b$ and $c ::_1 d$, meaning "b is a primary class of a" and "d is an immediate superclass of c." Each of these atoms can be model-theoretically interpreted as binary predicates.

In addition to the usual inference rules of F-logic, we also need axioms that state that primary class membership is a special kind of the usual class membership and that an immediate superclass is also a superclass in the old sense:

$$\begin{array}{l} X : Y \longleftarrow X :_1 Y \\ V :: W \longleftarrow V ::_1 W \\ V \neq W \longleftarrow V ::_1 W \end{array} \tag{32}$$

The following axioms say that immediate superclasses of the same class are incomparable in the class hierarchy and that an object may have at most one primary class.

$$\begin{array}{l} V' \doteq V'' \longleftarrow V ::_1 V' \wedge V ::_1 V'' \wedge V' :: V'' \\ X \doteq Y \longleftarrow Z :_1 X \wedge Z :_1 Y \end{array}$$

In addition, we need to ensure that class membership is a consequence of membership in a primary class, *i.e.*, that $a : b$ if and only if $a :_1 c$ and $c :: b$, for some c. This can be done by restricting F-programs so that is-a atoms, $X : Y$ and $V :: W$, will not occur in the rule-heads, except for the axioms (32). This also ensures that every subclass-relationship is a finite composition of the immediate subclass-relation.

16.6 Version Control

Although version control is not an intrinsically object-oriented feature, it has been often associated with object-oriented databases because of the importance of versioning in several "classical" object-oriented applications, such as computer aided design. Conceptually, version control has two distinct functionalities: creation of new versions and navigation among existing versions. As an example, we outline one of the several possible schemes for realizing versions in F-logic.

Suppose Mary and John are working cooperatively on a computer chip, denoted *chip*123. Each time Mary makes a modification to a version, v, of the chip, she creates a new object, $version(mary, v)$. Similarly, John constructs versions of the form $version(john, v)$. Thus, all versions of the chip have the form $version(p_n, \ldots version(p_1, chip123) \ldots)$, where each p_i is either *mary* or *john*.

Both Mary and John access the current version of *chip*123 via the name $current(chip123)$. To turn a version, say $version(mary, version(john, version(john, chip123)))$, into a current one, they can assert an equation of the form

$$current(chip123) \doteq version(mary, version(john, version(john, chip123)))$$

and delete any other equation of this form that might have been previously asserted.

Navigation through versions is quite easy. For instance, to access a previous version of the chip, one can use the following query:

$$?- \ (current(chip123) \doteq version(Whoever, PrevVersion)) \ \wedge \ PrevVersion\,[\ldots] \wedge \ldots$$

Another way to organize versions in F-logic is to define a method, *version*, that returns appropriate versions, depending on the argument. To access a version of *chip*123, one could use

$$?- \ chip123[version\,@arg \rightarrow Version] \wedge Version\,[\ldots]$$

where *arg* is a suitable encoding of the desired version. Yet another way to do versioning is to parameterize attributes with version id's, as suggested in [52].

17 Extensions to F-logic

In this section, we outline several extensions to F-logic. Some of these could have been included in the main text, but they were left out to help us focus on the main issues. These extensions include sorted F-logic and multisets. Other extensions discussed here are combining F-logic with HiLog [34], and modeling methods with side effects.

17.1 Sorted F-logic and its Uses

Sometimes it may be necessary to distinguish between set-valued and scalar attributes at the syntactic level. (Previously, we could distinguish them only at the meta-level, via the notion of type correctness). At other times, we may want to designate certain objects as "true" individual objects, namely, objects that can never play the role of a class. For instance, objects that represent people, such as *john*, *mary*, etc., may be in this category. In yet other cases, the user may prefer to distinguish oid's that represent "atomic values," such as integers or strings of characters, from oid's that represent objects with complex internal structure (such as employees or companies).

Using sorts to separate things that are not supposed to mingle in one domain is an old idea. The advantage of having sorts in addition to types is that well-formedness with respect to sorts is easy to verify.

To illustrate how this idea can be applied to F-logic, suppose we wanted to keep a distinction between classes and individuals and yet be able to manipulate both kinds of objects in a uniform way. To this end, we could separate all variables and function symbols into three categories with different name spaces. For instance, to represent individuals, we could use id-terms whose outermost symbol's name begins with a "!"; to represent classes, we may use id-terms beginning with a "$\sharp$"-symbol; and id-terms that begin with any other legal symbol (*e.g.*, a letter) can be used to represent *any* object, an individual or a class. These latter symbols provide access to both kinds of objects uniformly, making it possible to talk about both sorts together.

Semantically, sorts are accommodated in a standard way. The domain, U, of an F-structure would now consist of two disjoint parts: $U^{\sharp}$—to interpret class-objects; and $U^{!}$—for individual objects. Variables beginning with a $\sharp$-symbol will then range over the subdomain of classes, while !-variables will be restricted to the domain of individuals. All other variables will be allowed to range over the entire domain, U. Since we wish to allow class-objects to be constructed using both, class-objects and individual-objects, a function symbol beginning with a "$\sharp$" will have the type $U^n \mapsto U^{\sharp}$, for a suitable n, and a function symbol beginning with a "!"-symbol will have the type $U^n \mapsto U^{!}$.

Alternatively, the same effect can be achieved by introducing a pair of new unary predicates, $class(X)$ and $individual(X)$, where *class* and *individual* are true precisely of those ground id-terms whose outermost symbol starts with "$\sharp$" and "!", respectively. In addition, rules and queries should be relativized as follows: For every individual id-term, $!T$, in the body of a rule, $head \longleftarrow body$ (or in a query $?-\ query$), add $individual(!T)$ as a conjunct, obtaining $head \longleftarrow body \wedge individual(!T)$ (resp., $?-\ query \wedge individual(!T)$). Likewise, for every id-term that represents a class, $\sharp S$, add $class(\sharp S)$ as a conjunct in the body of the rule or the query.

Next, we may want to restrict logical formulas further. For instance, in $T\,[Mthd\,@\,V_1, \ldots, V_n \Rightarrow W]$, we may insist that T, $V_1, \ldots,$ V_n, W should represent classes, (*i.e.*, that all signatures will look like $\sharp T\,[Mthd\,@\,\sharp V_1, \ldots, \sharp V_n \Rightarrow \sharp W]$). Likewise, for is-a assertions of the form $O : Cl$ and $Cl :: Cl'$ we may require that Cl and Cl' will be classes.

The aforementioned dichotomy between atomic values and object ids can be introduced along similar lines: Object id's that denote atomic values (*e.g.*, integers) can be assigned to a distinct logical sort and placed in a separate subdomain. Then, the sorted language can be restricted to ensure that atomic values do not occur in the role of an oid in an object-molecule or in the role of a class in an is-a assertion.

The utility of sorts is not limited to distinguishing individuals, values, or methods from other objects. In principle, users may want to define their own naming conventions. In each case, this will add a new sort of symbols to the logic, to harbor the oid's of objects the user believes are worth distinguishing. We shall not discuss these language-engineering issues further, as they are beyond the scope of this paper.

17.2 Multiset-valued Methods

In recent years, the *multiset* data type has attracted much attention in database community. A multiset is a set where each element has a finite count that indicates the "number of times" the element is a member in the set.

Relations over multisets can be defined analogously to standard relations over sets. For instance, $e \in M$ would mean that e is a member of the multiset M at least once, while $e \in^k M$ would mean that e is in M at least k times. The sub-multiset relation, $M_1 \dot{\subseteq} M_2$, holds if, for every $k > 0$, each k-member of M_1 is also a member of M_2 at least k times.

Incorporation of multisets into F-logic is fairly easy. The idea is to allow set-valued methods to accept multisets as values, which effectively turns them into *multiset methods.*[31] To this end, we can introduce *counted id-terms*, which are expressions of the form $k \cdot T$, where T is the usual id-term and k is a natural number. These terms can occur to the right of "$\twoheadrightarrow$" and "$\bullet\!\!\twoheadrightarrow$" in set-valued method-expressions. For instance, $o\,[attr \twoheadrightarrow 3 \cdot e]$ means that e is a triple-member of the multiset returned by the attribute $attr$ on object o. Our old notation, $attr \twoheadrightarrow t$, is then treated as a syntactic sugar for $attr \twoheadrightarrow 1 \cdot t$.

The semantics of F-molecules with multiset methods needs only a few modifications. First, the mappings $I_{\twoheadrightarrow}$ and $I_{\bullet\!\twoheadrightarrow}$ will now have to return multisets. Second, in Definition 5.1(iii) (molecular satisfaction for set-valued methods), the term "containment" should be understood in the sense of multiset containment. That is, we shall now write

$$\mathbf{I} \models_\nu O\,[SetM\,@\,R_1, \ldots, R_l \twoheadrightarrow \{S_1, \ldots, S_m\}]$$

where $S_1, \ldots, S_m$ are counted id-terms, if and only if the multiset $I^{(l)}_{\twoheadrightarrow}(\,\nu(SetM)\,)(\nu(O), \nu(R_1), \ldots, \nu(R_l))$ exists and contains the multiset $\{\nu(S_1), \ldots, \nu(S_m)\}$.[32]

For the proof theory, we only need to take care of multiset unification. For instance, we would like the molecule $o\,[attr \twoheadrightarrow 2 \cdot X]$ to unify into $o\,[attr \twoheadrightarrow 3 \cdot Y]$, but not into $o\,[attr \twoheadrightarrow X]$. This is achieved by properly defining the notion of *submolecule* in Definition 11.1. More precisely, rather than changing the notion of submolecule, we need to adjust the concept of *constituent atoms* (defined at the end of Section 7.4) to handle multisets properly. For this, we can postulate that if α is a constituent atom of a molecule, β,

[31] Alternatively, we could introduce multiset methods as being distinct from set-valued methods.

[32] For a counted id-term, $\nu(k \cdot T)$ stands for $k \cdot \nu(T)$.

then any atom that is the same as α, but has a lower membership count, is also a constituent atom of β. For instance, both $o[attr \twoheadrightarrow X]$ and $o[attr \twoheadrightarrow 2 \cdot Y]$ are constituent atoms of $o[attr \twoheadrightarrow \{2 \cdot X,\, 2 \cdot Y\}]$. The rest of the definitions need no change.

17.3 HiLog-inspired Extensions

HiLog [34] is a higher-order extension of predicate calculus that allows variables to range over names of function and predicate symbols. Furthermore, these names themselves may have a rather complex structure and, in particular, the usual first-order terms (with variables) may appear in predicate positions. Many applications of HiLog are described in [34, 110, 92] and it is clear that extending F-logic in this direction is a good idea. In Section 12.4.1, we have seen one example—graph restructuring—where a combination of HiLog and F-logic may be useful.

We illustrate HiLog using the following example:

$$\begin{array}{l} closure(Graph)(From, To) \longleftarrow Graph(From, To) \\ closure(Graph)(From, To) \longleftarrow Graph(From, Mid) \wedge closure(Graph)(Mid, To) \end{array}$$

This HiLog program computes the closure of any binary relation that is passed to the predicate constructor *closure* as an argument. Here *Graph* is a variable that ranges over predicates and *closure*(*Graph*) is a *non-ground* term that denotes a parameterized family of predicates. For any given binary predicate, *graph*, the term *closure*(*graph*) is another binary predicate that is true of a pair $\langle from, to \rangle$ precisely when *graph* has a directed path connecting *from* and *to*.

The ideas underlying HiLog are applicable to a wide range of logical languages, and the reader familiar with HiLog will have no difficulty to integrate them into F-logic by providing a semantics to HiLog-style terms. To this end, all that needs to be done is the change the interpretation of id-terms from the one given in this paper to that in [34].

17.4 F-logic and Dynamic Behavior

F-logic does not cover one important aspect of object-oriented languages—dynamic behavior. That is, in F-logic, one can define methods for querying object's internal state, but the logic provides no means for defining methods to actually change this state.

Recently, a new logical theory, called *Transaction Logic*, was proposed in [21, 22, 23]. Although formulated in terms of classical predicate logic, the ideas underlying Transaction Logic are equally applicable to F-logic, extending it with dynamic behavior. While we cannot elaborate on Transaction Logic in this paper, an example, below, shows one way in which Transaction Logic and F-logic can be combined. This example also illustrates a style of programming that is not possible in F-logic alone.

Consider the power-set example (13 – 15) of Section 12.4.1. Suppose s is a member of the class *set*, and let us assume that the following fact is true: $s[self \twoheadrightarrow \{a, b, c\}]$. It is easy to see that the power-set of s (as defined by the rules in (13 – 15)) is as follows:

$$\begin{array}{ll} s[powerset \twoheadrightarrow \{ & \emptyset,\ add(a,\emptyset),\ add(b,\emptyset),\ add(c,\emptyset),\ add(a, add(b,\emptyset)), \\ & add(b, add(c,\emptyset)), add(a, add(c,\emptyset)) add(a, add(b, add(c,\emptyset)))\ \}] \end{array}$$

The oid's inside the braces denote different subsets of the set represented by s. However, on close examination, the reader may discover that each such oid duplicates the contents of the entire set it represents! For instance, the following F-molecule is entailed by the power-set program:

$$add(b, add(c, \emptyset))[self \twoheadrightarrow \{b, c\}]$$

Here, $add(b, add(c, \emptyset))$ represents the set $\{b, c\}$ whose members occur in the set returned by the attribute *self* and also in the oid itself. Such duplication of b and c is hardly desirable. A more natural way would be to "invent" a new oid each time a new subset of s is computed.

Unfortunately, there is no natural way to do oid-invention in F-logic. Other languages, such as IQL [3], introduce oid-invention as a basic mechanism for defining new objects. However, while IQL's syntax is motivated by Datalog, IQL is not based on a logic. In contrast, Transaction Logic provides a logical basis for oid invention (and many other aspects of database dynamics).

To illustrate how power-sets can be defined in a combined language of F-logic and Transaction Logic, we introduce a predicate, *new_id(X)*, that binds X to a new constant each time this predicate is "evaluated." A novice to Transaction Logic should accept—on faith—that there is a logical way (in fact, several ways) to achieve the effect of binding X to a new value on each successful evaluation of $new\_id(X)$.

The program, below, constructs power-sets without duplicating the elements of the sets in the oids. The algorithm resembles the usual, procedural way of constructing power-sets while, at the same time, it bears strong similarity to the power-set program of Section 12.4.1. In building our program, we first define a new relation over sets, $added(Y, S, S')$, that represents the fact that the oid S' denotes a set obtained from the set denoted with S by adding the element Y (which may or may not already be in S). This is reflected in the following rule (assuming that no other rule defines the attribute *self* of S'):

$$S'[self \twoheadrightarrow \{X, Y\}] \longleftarrow S : set \wedge S' : set \wedge added(Y, S, S') \wedge S\,[self \twoheadrightarrow X] \tag{33}$$

To define the *powerset* attribute, we start by postulating that the empty set is a member of the power-set of every set:

$$S\,[powerset \twoheadrightarrow \emptyset\,] \longleftarrow S : set$$

The next step is to define a method (a *transaction* in the terminology of Transaction Logic) that creates a *new* subset of a given set by adding a single element to an already known subset:

$$\begin{array}{rcl} S\,[newSubset \rightarrow S'] & \longleftarrow & S : set\,[self \twoheadrightarrow Z;\ powerset \twoheadrightarrow S''] \qquad (34) \\ & & \otimes\, new\_id(S') \otimes added.ins(Z, S'', S') \\ & & \otimes \neg oldSubset(S', S) \otimes S\,[powerset.ins \twoheadrightarrow S'] \\ oldSubset(S', S) & \longleftarrow & S\,[powerset \twoheadrightarrow S''] \wedge setEqual(S'', S') \qquad (35) \end{array}$$

The first rule here says, to construct a new subset of S (one that is not yet in the set determined by the attribute *powerset*), first find an already known subset of S, say, S''. Then generate a new id, S', using the "dynamic" predicate *new_id* mentioned earlier. Next, (34) inserts $added(Z, S'', S')$ to record the fact that S' is obtained from S'' by adding Z (which is represented by the predicate *added.ins*—see [21, 23]). If S' does not represent an already computed subset of S, then insert the fact that S' is a newly computed element of the power-set of S. The actual insertions take place when the predicate *added.ins* and the F-atom $S\,[powerset.ins \twoheadrightarrow S']$ are evaluated. The first insertion is conditional; it depends on the validity of the post-condition, $\neg oldSubset(S', S)$, which prunes away the wrong choices for S'' in (34)—choices where S' is one of the already computed (*i.e.*, old) subsets of S.

Observe that we do not define the new set, S', explicitly. Instead, by inserting $added(Z, S'', S)$ into the database, the attribute *self* of S' gets its new value via the rule (33).

The symbol "$\otimes$" in (34) is a new logical connective in Transaction Logic, called *serial conjunction.* Informally, $a \otimes b$ says, "do a, then do b." An important point here is that the method $newSubset$ has a "side-effect." That is, in the process of computation, the database state changes from the initial state to one in which $added(Z, S'', S')$ is true for some values of Z, S'', and S' (and nothing else changes); then it is further changed into a state where, in addition, $S\,[powerset \twoheadrightarrow S']$ becomes true. The logic behind such executions is given in [21].

Rule (35) above says that an object, S', represents an "old" subset of S if it represents a set of objects that is set-equal to some other object already in the *powerset* of S. Set-equality is verified via the predicate *setEqual*, which is analogous to a predicate defined earlier, in (12) of Section 12.4.1.

Finally, we define a transaction that does the actual job of building power-sets:[33]

$$\begin{array}{lcl} build\_powerset(S) & \longleftarrow & S\,[newSubset \twoheadrightarrow S'] \otimes build\_powerset(S) \\ build\_powerset(S) & \longleftarrow & \Box \neg S\,[newSubset \twoheadrightarrow S'] \end{array}$$

The first rule here says, if a new subset of S can be constructed, then keep going. The second rule is a termination condition. It says that the transaction *build_powerset(S)* succeeds if it is no longer possible to construct a new subset of S. The operator "$\Box$" here is the *necessity operator* of Transaction Logic; it is defined in [21].

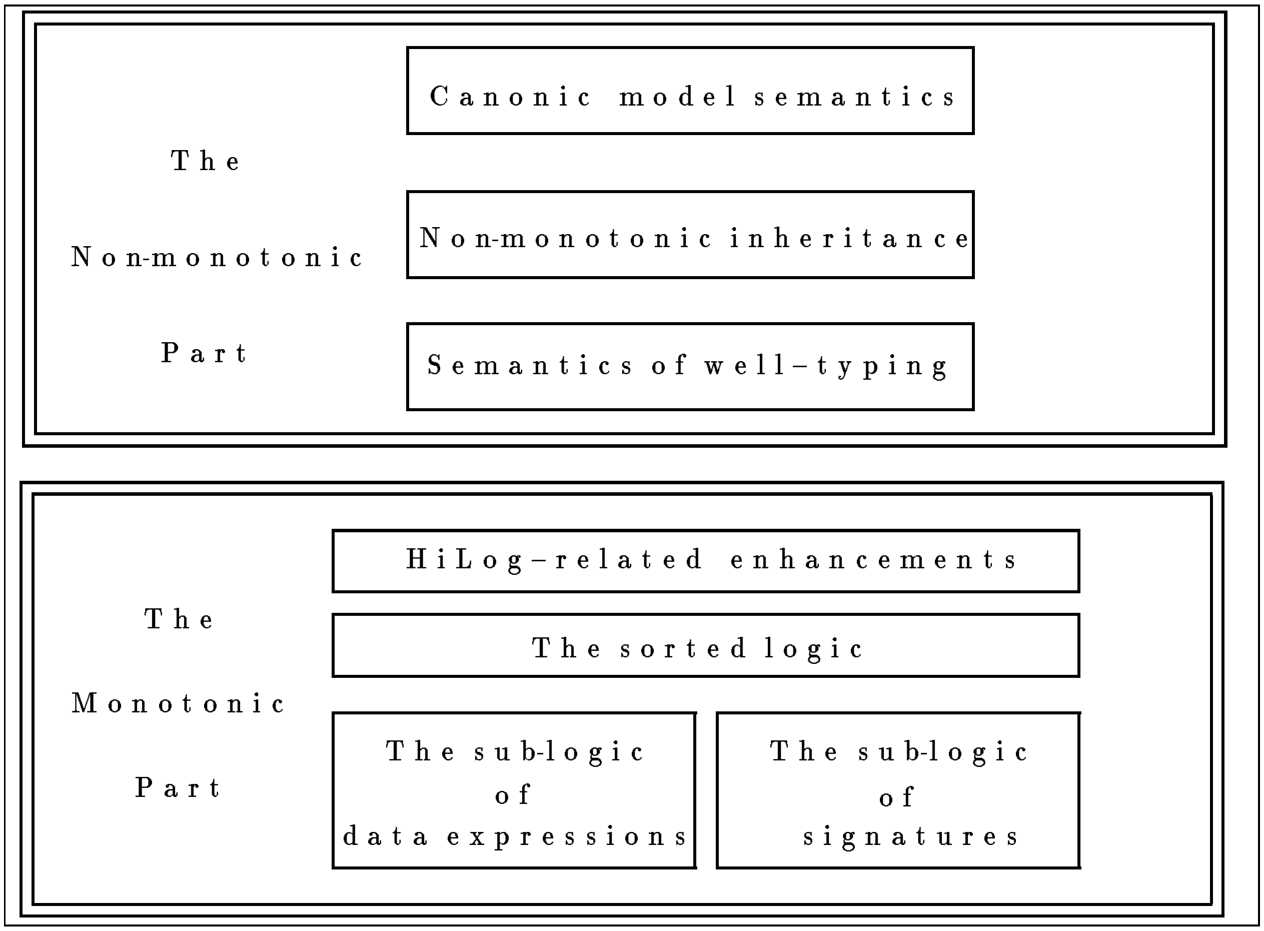
18 The Anatomy of F-logic

Having survived a heavy barrage of definitions, it must be gratifying to get an opportunity to take a retrospective look at the structure of F-logic and clarify the various relationships that exist among its different components. A careful examination would then reveal that many components are independent from each other, both semantically and proof-theoretically. A high-level view of the internals of F-logic is depicted in Figure 9. The logic is presented as a bag of ideas that can be classified into two main categories. The monotonic part of F-logic was presented in Sections 4 through Section 11. The non-monotonic part is comprised of a number of techniques, all based on the canonic model semantics. These techniques are described in Sections 12, 13, 15, and in Appendices A and B. The non-monotonic part of F-logic is based on the monotonic part, especially on its model theory. The proof theory presented in Section 11 is sound and complete for the monotonic part. The non-monotonic extensions do not possess a complete proof theory (without making strong assumptions about the syntax), but sound evaluation strategies can be developed based on the monotonic proof theory. This is analogous to classical logic, where logic programs with stratified negation have no complete proof theory.

The two main components of the monotonic department are the logics of data expressions and the logic of signatures. These sublogics are independent from each other, as can be seen by inspecting the semantics and the proof theory presented earlier. The glue between these two sublogics is provided by the well-typing semantic conditions of Section 13, which belong to the non-monotonic part of the logic. Furthermore, notice that, as far as the monotonic logic is concerned, inheritable and non-inheritable data expressions are completely independent, and, in fact, their properties are virtually identical. The difference can be seen only when it comes to typing and non-monotonic inheritance.

The third component is a technique for defining sorts in F-logic, discussed in Section 17.1. The fourth component consists of the ideas borrowed from HiLog [34]. This extends the syntax of F-logic by

[33] Alternatively, we could define *powerset* as a *method with side effects* by writing $S\,[build\_powerset \rightarrow null]$ instead of $build\_powerset(S)$. Here "null" is an oid that we designated to represent methods that return no answers.



The Non-monotonic Part
Canonic model semantics
Non-monotonic inheritance
Semantics of well–typing
The Monotonic Part
HiLog–related enhancements
The sorted logic
The sub-logic of data expressions
The sub-logic of signatures

Figure 9: The Internal Structure of F-logic

allowing variables and even complex terms to appear in the positions of predicate and function symbols, as is discussed in Section 17.3.

The non-monotonic part of F-logic is also composed of a collection of distinct ideas. One of them is the canonic model semantics for F-programs with equality, which is discussed in Section 12 and in Appendix A. The next two determine the semantics of non-monotonic inheritance and typing, both based on the idea of a canonic model. The notion of encapsulation also belongs here, since we view it simply as an elaborate policy of type-correctness. Note that, although Appendix A defines perfect models as an example of a canonic model semantics, our treatment of inheritance and typing is largely independent from this specific theory of canonic models.

Because of the independence of its various components, F-logic can be viewed as a tool-kit for building customized deductive object-oriented languages. In fact, there is a finer division within the logic. For instance, the semantics of scalar methods is independent from that of set-valued methods, and we can consider each type of methods as a separate tool in the kit. Likewise, we could require the oid's of classes and methods to be constants; method overloading can also be controlled in a number of ways (using sorts, for example). This tool-kit-like structure of F-logic is extremely helpful, as it lets one address different aspects of the logic in separation from each other, both theoretically and implementationally.

The last issue we would like to discuss is the relationship between predicate calculus and F-logic. In one direction, predicate calculus is a subset of F-logic and so the latter may seem to be more powerful than the former. However, it turns out that F-logic can be encoded in classical logic:

Theorem 18.1 *There are mappings*

$$\begin{array}{lll} \Gamma & : \{\textit{F-formulas}\} & \longrightarrow \ \{\textit{well-formed formulas of predicate calculus}\} \\ \Phi & : \{\textit{F-structures}\} & \longrightarrow \ \{\textit{semantic structures of predicate calculus}\} \end{array}$$

such that

$$\mathbf{M} \models_F \psi \quad \textit{if and only if} \quad \Phi(\mathbf{M}) \models_{PC} \Gamma(\psi) \tag{36}$$

for any F-structure $\mathbf{M}$ *and any F-formula* ψ*, where "*$\models_F$*" and "*$\models_{PC}$*" denote logical entailment in F-logic and predicate calculus, respectively.* □

Proof: (Sketch) The proof is easy but tedious, so most of it is left as an exercise. The main idea is to introduce *new* predicates that encode the meaning of the methods for every arity and for every type of invocation.

Let $scalarNonInheritable_n$ be a new predicate that we shall use to encode n-ary non-inheritable scalar data expressions. Similarly, we shall use $setNonInheritable_n$ to encode non-inheritable set-valued data expressions. Then we can split F-molecules into constituent atoms and represent each atom by an appropriate tuple in one of the above predicates. In addition, we shall need predicates $scalarType_n$ and $setType_n$ to encode scalar and set-valued signature expressions. For instance,

$$bob\,[jointWorks\,@phil \twoheadrightarrow X] \wedge empl\,[budget \Rightarrow integer]$$

can be represented as

$$setNonInheritable_1(jointWorks, bob, phil, X) \wedge scalarType_0(budget, empl, int)$$

Class membership and subclassing can be represented via appropriate binary predicates. To complete the encoding Γ, we will have to specify axioms that achieve the effect of the built-in features of F-structures. For instance, it will be necessary to write down transitivity axioms for subclassing, type inheritance axioms, etc.

The semantic mapping Φ can be defined along similar lines. Once the mappings Γ and Φ are defined in this way, verifying (36) becomes a simple, albeit lengthy, task. Examples of this kind of proofs can be found in [108, 34]. □

The idea that object-based logics can be encoded in classical predicate logic first appeared in [35], but embeddings of this sort were known in the Prolog community for quite some time. For instance, some prologs provide syntactic sugar that, roughly, amounts to the incorporation of what we earlier called "complex values." A similar transformation was also used in [76] to extend Prolog with object-style syntax.

The reader whose faith was shaken by the fact that F-logic is, in a sense, equivalent to predicate calculus, may find comfort in the following arguments. First, semantics-by-encoding, as in Theorem 18.1, is inadequate, as it is indirect and provides no insight to the user when it comes to understanding the meaning of a program. Since our goal is to provide a logical rendition for a class of languages that are collectively classified as object-oriented, taking the mapping Γ of Theorem 18.1 for the meaning of

F-logic would be a misnomer. Indeed, Γ is merely an algorithm that sheds little light on the nature of object-oriented concepts the logic is designed to model. The proof sketch of Theorem 18.1 should make it clear that even for simple F-programs their Γ-image is not easily understandable. Another argument—which is well articulated in the concluding section to [34]—can be summarized as follows: The syntax of a programming language is of great importance, as it shapes the way programmers approach and solve problems. However, syntax without direct semantics is not conducive to programming. Third, a direct semantics for a logic suggests ways of defining proof theories tailored to that logic. Such proof theories are likely to be a better basis for implementation than the general-purpose classical proof theory. Lastly, Theorem 18.1 relates only the monotonic part of F-logic to classical logic. Mapping the non-monotonic components of F-logic into a non-monotonic theory for predicate calculus does not seem to lead to a useful logic.

19 Conclusion

Unlike the relational approach that was based on theoretical grounds from the very beginning, the object-oriented approach to databases was dominated by "grass-roots" activity where several systems were built without the accompanying theoretical progress. As a result, many researchers felt that the whole area of object-oriented databases is misguided, lacking direction and needing a spokesman, like Codd, who could "*coerce the researchers in this area into using a common set of terms and defining a common goal that they are hoping to achieve* [86]."

Our contention is that the problem lies deeper than that. When Codd made his influential proposal, he was relying on a large body of knowledge in classical predicate logic. He had the insight to see a very practical twist to rather theoretical ideas in mathematical logic, which led him to develop a theory that revolutionized the entire database field. Until now, logical foundations for object-oriented databases that are parallel to those underlying the relational theory were lacking and this was a major factor for the uneasy feeling. In a pioneering work [73], Maier proposed a framework for defining a model-theoretic semantics for a logic with an object-oriented syntax. However, he encountered many difficulties with this approach and subsequently abandoned this promising direction. As it turned out, the difficulties were not insurmountable, and the theory was repaired and significantly extended in [35, 58].

In this paper, we proposed a novel logic that takes the works reported in [35, 58, 60] to a new dimension: F-logic is capable of representing virtually all aspects of what is known as the object-oriented paradigm. We provided a formal semantics for the logic and showed that it naturally embodies the notions of complex objects, inheritance, methods, and types. F-logic has a sound and complete resolution-based proof procedure, which makes it also computationally attractive and renders it a suitable basis for developing a theory of object-oriented logic programming.

F-logic is also an *extensible* logic, as it can be combined with other recently proposed logics for knowledge representation, such as HiLog [34], Transaction Logic [21], and Annotated Predicate Logic [20, 56, 57]. Two such combinations were sketched in Section 17. This extensibility puts F-logic in the center of an emerging unified logical formalism for knowledge and data representation.

Acknowledgments: We are indebted to Catriel Beeri for his very detailed comments whose length rivals the length of this paper. Many other improvements are due to the suggestions of Rodney Topor. We are grateful to Alexandre Lefebvre and Heinz Uphoff for pointing out a problem with an earlier version of the semantics for inheritance. We also wish to thank Serge Abiteboul, Mira Balaban, Joachim Biskup, Weidong Chen, Stephen Hegner, Rick Hull, David Maier, Sergio Martin, Ken Ross, Shuky Sagiv, Moshe Vardi, David Warren and Gio Wiederhold for discussions that helped shape this paper. Last, but not least, thanks goes to the anonymous referees for their constructive critique.

A Appendix: A Perfect-Model Semantics for F-logic

To adapt the various semantics for negation (such as [89, 46, 105, 104, 90, 5, 62]) to F-logic, the general principle is to use method names in contexts where predicates are used in the classical setting. For the perfect-model semantics [89], this means that stratification has to be ensured with respect to method names, so the program

$$X\,[wants \twoheadrightarrow Y] \longleftarrow \neg X\,[has \twoheadrightarrow Y]$$

would be stratified despite the recursion through negation within the same object, X. Note that, if x, y are ground terms, $\neg x\,[has \twoheadrightarrow y]$ means that y is not in the set $x.has$, the value of the attribute *has* on x; it does *not* mean that $x.has$ is empty or undefined. In contrast, consider the following one-rule program (adapted from [105]):

$$\begin{array}{rl} sillyGame\,[winningPos \twoheadrightarrow Pos_1] \longleftarrow & \neg sillyGame\,[winningPos \twoheadrightarrow Pos_2] \\ & \wedge\, Pos_1[legalMoves \twoheadrightarrow Pos_2] \end{array}$$

It is not locally stratified in F-logic since there is recursion through negation in the attribute *winningPos* when both Pos_1 and Pos_2 are instantiated to the same constant, say *badPos*.

However, the process of adapting perfect models to F-logic is more involved than this discussion may suggest. We present a solution in two steps. First, we apply the above general principle directly and come up with a preliminary definition. Then we point out some problems with this definition and show how they can be corrected.

Preliminary Definitions

Let $\mathcal{L}$ be an F-language and $\mathbf{P}$ be a general F-program. We define a *dependency graph*, $\mathcal{D}(\mathcal{L}, \mathbf{P})$, as follows. Let $\mathbf{P}^*$ denote the set of all ground instances of the rules in $\mathbf{P}$. The *nodes* of $\mathcal{D}(\mathcal{L}, \mathbf{P})$ correspond to ground atoms in the Herbrand base, $\mathcal{HB}(\mathcal{L})$. A *positive arc*, $\varphi \overset{\oplus}{\longleftarrow} \psi$, connects a pair of nodes, φ and ψ, if there is a rule, $\bar{\varphi} \longleftarrow \cdots \wedge \bar{\psi} \wedge \cdots$, in $\mathbf{P}^*$ such that φ and ψ are constituent atoms of $\bar{\varphi}$ and $\bar{\psi}$, respectively. A *negative arc*, $\varphi \overset{\ominus}{\longleftarrow} \psi$, is in $\mathcal{D}(\mathcal{L}, \mathbf{P})$ when $\mathbf{P}^*$ has a rule of the form $\bar{\varphi} \longleftarrow \cdots \wedge \neg\bar{\psi} \wedge \cdots$. It is possible for a pair of nodes to be connected by a positive and a negative arc at the same time.

We shall write $\phi \preceq_{\mathbf{P}} \psi$, where ϕ and ψ are atoms in $\mathcal{HB}(\mathcal{L})$, if $\mathcal{D}(\mathcal{L}, \mathbf{P})$ has a directed path from ϕ to ψ. We write $\phi \prec_{\mathbf{P}} \psi$ if there is a *negative path* from ϕ to ψ (*i.e.*, if some arcs on the path are negative). In general, $\prec_{\mathbf{P}}$ and $\preceq_{\mathbf{P}}$ are not partial orders. However, $\prec_{\mathbf{P}}$ is a partial order when $\mathcal{D}(\mathcal{L}, \mathbf{P})$ has no cycles that contain negative edges, the *negative cycles.*

Definition A.1 (Locally Stratified Programs, preliminary) An F-program, $\mathbf{P}$, is *locally stratified* if the relation $\prec_{\mathbf{P}}$ is *well-founded*, *i.e.*, if $\mathcal{D}(\mathcal{L}, \mathbf{P})$ has no infinite decreasing chains of the form $\cdots \prec_{\mathbf{P}} \phi_2 \prec_{\mathbf{P}} \phi_1$. □

This definition implies that, for locally stratified programs, $\prec_{\mathbf{P}}$ is irreflexive and asymmetric (for, otherwise, if $\phi_1 \prec_{\mathbf{P}} \phi_2$ and $\phi_2 \prec_{\mathbf{P}} \phi_1$, for some ϕ_1 and ϕ_2, there is an infinite chain $\cdots \prec_{\mathbf{P}} \phi_1 \prec_{\mathbf{P}} \phi_2 \prec_{\mathbf{P}} \phi_1$).

Similarly, Definition A.1 implies that the graph $\mathcal{D}(\mathcal{L}, \mathbf{P})$ does not have negative cycles when $\mathbf{P}$ is locally stratified.

We can now introduce a *preference* quasi-order on H-structures as follows: an H-structure $\mathbf{M}$ is *preferable* to $\mathbf{L}$, denoted $\mathbf{M} \ll_{\mathbf{P}} \mathbf{L}$, if whenever there is an atom $\varphi \in \mathbf{M} - \mathbf{L}$ then there is an atom $\psi \in \mathbf{L} - \mathbf{M}$ such that $\psi \prec_{\mathbf{P}} \varphi$.

Definition A.2 (Perfect Models, preliminary) Let $\mathbf{P}$ be a locally stratified F-program and $\ll_{\mathbf{P}}$ be the associated quasi-order on its H-models. H-models of $\mathbf{P}$ that are minimal with respect to $\ll_{\mathbf{P}}$ are called *perfect* models of $\mathbf{P}$. $\square$

In general, $\ll_{\mathbf{P}}$ is a quasi-order, not a partial order. However, as in [89], it can be shown that if $\mathbf{P}$ is locally stratified, $\ll_{\mathbf{P}}$ is a partial order on the H-models of $\mathbf{P}$. Also, as in the classical theory, it can be shown that every locally stratified program has a unique perfect model.

To provide a more direct characterization of perfect models, we need to define *program stratification.* To this end, we extend the pre-orders $\preceq_{\mathbf{P}}$ and $\prec_{\mathbf{P}}$ from atoms in $\mathcal{HB}(\mathcal{L})$ to rules in $\mathbf{P}^*$. This is conveniently done with the help of the rule-dependency graph, $\mathcal{D}_r(\mathcal{L}, \mathbf{P})$, which contains more information than the plain dependency graph, $\mathcal{D}(\mathcal{L}, \mathbf{P})$. This graph is also needed in Appendix B.

A *rule-dependency graph*, $\mathcal{D}_r(\mathcal{L}, \mathbf{P})$, is a graph whose nodes are the rules in $\mathbf{P}^*$, i.e., all the ground instances of the rules in $\mathbf{P}$. A *positive arc*, $r \xleftarrow{\oplus} r'$, where $r, r' \in \mathbf{P}^*$, exists if, and only if, a constituent atom of the molecule in the head of r' occurs also in a molecule in the body of r. A *negative arc*, $r \xleftarrow{\ominus} r'$, exists if a constituent atom of the molecule in the head of r' *occurs negatively* (*i.e.*, with a $\neg$-sign) in a molecule in the body of r.

The pre-orders $\preceq_{\mathbf{P}}$ and $\prec_{\mathbf{P}}$ are now defined on $\mathbf{P}^*$ the same way they were defined on $\mathcal{HB}(\mathcal{L})$, except that the graph $\mathcal{D}_r(\mathcal{L}, \mathbf{P})$ is now being used instead of $\mathcal{D}(\mathcal{L}, \mathbf{P})$: if $C_1, C_2 \in \mathbf{P}^*$ then $C_1 \preceq_{\mathbf{P}} C_2$ (or $C_1 \prec_{\mathbf{P}} C_2$) if $\mathcal{D}_r(\mathcal{L}, \mathbf{P})$ has a path (resp., a negative path) from C_1 to C_2.

Given a locally stratified program, $\mathbf{P}$, we can partition the rules in $\mathbf{P}^*$ into *stratas*, $\mathbf{P}_1^*$, $\mathbf{P}_2^*$, ..., in accordance with the priority relations $\prec_{\mathbf{P}}$ and $\preceq_{\mathbf{P}}$, so that:

$$\text{If } C_1 \in \mathbf{P}_i^*,\ C_2 \in \mathbf{P}_j^* \text{ and } C_1 \preceq_{\mathbf{P}} C_2, \text{ then } i \le j;\ \text{ if } C_1 \prec_{\mathbf{P}} C_2 \text{ then } i < j. \tag{37}$$

Partitions that satisfy (37) are called *stratifications.*

As in the classical theory, it can be shown that the perfect model of $\mathbf{P}$ can be computed by firing rules of $\mathbf{P}^*$ in the stratification order. Let $T_{\mathbf{P}}$ be a *generalized one-step inference operator* defined as follows:

$$\begin{array}{ll} \{\mathit{head} \mid & \mathit{head} \longleftarrow l_1 \wedge \cdots \wedge l_n \wedge \neg l_{n+1} \wedge \neg l_{n+m} \text{ is a rule in } \mathbf{P}^*, \\ & \text{such that } l_1, \ldots, l_n \in \mathbf{I} \text{ and } l_{n+1}, \ldots, l_{n+m} \notin \mathbf{I}\,\} \end{array}$$

Then $\mathbf{M}$, the perfect model of $\mathbf{P}$, can be computed as follows:

$$\begin{array}{lcl} \mathbf{M}_1 & = & T_{\mathbf{P}_1^*} \uparrow \omega\,(\emptyset) \\ \mathbf{M}_2 & = & T_{\mathbf{P}_2^*} \uparrow \omega\,(\mathbf{M}_1) \bigcup \mathbf{M}_1 \\ \mathbf{M}_3 & = & T_{\mathbf{P}_3^*} \uparrow \omega\,(\mathbf{M}_2) \bigcup \mathbf{M}_2 \\ & \ddots & \\ \mathbf{M} & = & \bigcup_{i=1}^{\infty} \mathbf{M}_i \end{array} \tag{38}$$

As in the classical case, the result of the above iteration is independent of the stratification.

Revised Definitions

Unfortunately, the above straightforward adaptation of classical perfect models to F-logic is inadequate. One problem arises due to the equality predicate, " $\doteq$ ". In [89], perfect models were introduced assuming the so called *freeness* axioms for equality, which postulate that ground terms are equal if and only if they are identical. Without these axioms, perfect models do not provide natural semantics even for simple locally stratified programs. For instance, consider the following pair of F-programs:

$$\mathbf{P}_1: \begin{array}{l} a \doteq b \\ p(a) \longleftarrow \neg p(b) \end{array} \qquad \mathbf{P}_2: \begin{array}{l} a \doteq b \longleftarrow \neg p(b) \\ p(a) \end{array} \tag{39}$$

The unique perfect model of $\mathbf{P}_1$ is $M_1 = \{p(a), p(b), a \doteq b\}$, and $M_2 = \{p(a), p(b)\}$ is the unique perfect model of $\mathbf{P}_2$. (These models are obtained using the definitions in [89] in a simple-minded way, *i.e.*, by doing minimization as if " $\doteq$ " were an ordinary predicate.) The trouble is that $p(b)$ in both models is not supported by any rule. In fact, in both programs, $p(b)$ was derived assuming $\neg p(b)$ is true.

In classical logic programming, the equality problem can be side-stepped by banishing " $\doteq$ " from the rule heads. Alas, this shortcut is not appropriate for F-logic, since equality may be derivable even if it is not mentioned explicitly. For instance, $\{a :: b, b :: a\} \models a \doteq b$ and $\{a\,[attr \rightarrow b], a\,[attr \rightarrow c]\} \models b \doteq c$.

Another difficulty comes from the closure properties inherent in F-programs. For instance, signature inheritance corresponds to the following deductive rule:

$$X\,[M \,@\, \overrightarrow{args} \Rightarrow Y] \longleftarrow X :: Z \wedge Z\,[M \,@\, \overrightarrow{args} \Rightarrow Y] \tag{40}$$

This rule can lead to a situation similar to recursive cycles through negation. To illustrate the problem, consider the following program:

$$\begin{array}{l} b :: a \\ p(a) \longleftarrow \neg b\,[attr \Rightarrow c] \\ a\,[attr \Rightarrow c] \longleftarrow p(a) \end{array} \tag{41}$$

By itself, (41) does not cause problems. However, together with (40), we obtain a negative cycle going from $\neg b\,[attr \Rightarrow c]$ to $p(a)$ to $a\,[attr \Rightarrow c]$ to $b\,[attr \Rightarrow c]$.

Fortunately, there is a simple way out of these semantic quandaries (which also works for classical logic programs with equality). The idea is to account for equality and built-in constructs of F-logic by augmenting $\mathbf{P}$ with additional axioms. Each of these new axioms, called *closure* axioms, corresponds to an inference rule of Section 11. The only inference rules that do not lead to new axioms are the rules of resolution, factorization, merging, and elimination.

Axioms of paramodulation. For every arity, n, and every predicate symbol, p (including $\doteq$), we have the following axioms, where "$\rightsquigarrow$" stands for either of the six arrow types:

$$\begin{array}{rcl} p(Y_1, \ldots, Y_i', \ldots, Y_n) & \longleftarrow & p(Y_1, \ldots, Y_i, \ldots, Y_n) \wedge (Y_i \doteq Y_i') \\ X\,[M' \,@\, A_1, \ldots, A_n \rightsquigarrow V] & \longleftarrow & X\,[M \,@\, A_1, \ldots, A_n \rightsquigarrow V] \wedge (M \doteq M') \end{array}$$

Similar rules are needed for paramodulating on X, V, and on each of the A_i. Notice that paramodulation also accounts for the transitivity and the symmetry of the equality predicate. Additionally, to account for the paramodulation into subterms, the following axiom schema is needed:

$$f(X_1, \ldots, X_i', \ldots, X_n) \doteq f(X_1, \ldots, X_i, \ldots, X_n) \longleftarrow X_i \doteq X_i'$$

for every function symbol, f, in the language.

Note that similar axioms are needed in the classical case, if we are to define perfect models in the presence of equality.

Axioms of the IS-A hierarchy.

$$\begin{array}{rll} X :: Y & \longleftarrow X :: Z \wedge Z :: Y & \text{\% Transitivity} \\ X \doteq Y & \longleftarrow X :: Y \wedge Y :: X & \text{\% Acyclicity} \\ X : Y & \longleftarrow X : Z \wedge Z :: Y & \text{\% Subclass inclusion} \end{array}$$

Axioms of typing. For every arity, n, and for $\approx\!>$ standing for either $\Rightarrow$ or $\Rightarrow\!\!>$:

$$\begin{array}{rcl} X\,[M@A_1,\ldots,A_n \approx\!> T] & \longleftarrow & X'[M@A_1,\ldots,A_n \approx\!> T] \wedge X :: X' \\ & & \qquad \text{\% Type inheritance} \\ X\,[M@A_1,\ldots,A_i,\ldots,A_n \approx\!> T] & \longleftarrow & X\,[M@A_1,\ldots,A_i',\ldots,A_n \approx\!> T] \wedge A_i :: A_i' \\ & & \qquad \text{\% Input restriction} \\ X\,[M@A_1,\ldots,A_n \approx\!> T] & \longleftarrow & X\,[M@A_1,\ldots,A_n \approx\!> T'] \wedge T' :: T \\ & & \qquad \text{\% Output relaxation} \end{array}$$

Axioms of scalarity. For every arity, n, and for $\leadsto$ standing for $\rightarrow$ or $\bullet\!\!\rightarrow$:

$$V \doteq V' \;\longleftarrow\; X\,[M@A_1,\ldots,A_n \leadsto V] \wedge X\,[M@A_1,\ldots,A_n \leadsto V']$$

Definition A.3 (Augmented Programs) For any program, $\mathbf{P}$, let its *augmentation*, $\mathbf{P}^a$, be the union of $\mathbf{P}$ with the *relevant* ground instances of the closure axioms.

A ground instance of a closure axiom is *relevant* if all atoms in its premises are relevant. An atom is relevant if it occurs in the head of a rule in $\mathbf{P}^*$ or, recursively, in the head of a axiom-instance whose relevance was established previously. □

We can now rectify our earlier definitions by considering the order $\prec_{\mathbf{P}^a}$ instead of $\prec_{\mathbf{P}}$, *i.e.*, by using augmented programs instead of the original programs.

Definition A.4 (Locally Stratified Programs) An F-program, $\mathbf{P}$, is *locally stratified* if the relation $\prec_{\mathbf{P}^a}$ is *well-founded.* □

Definition A.5 (Perfect Models) H-models of $\mathbf{P}$ that are minimal with respect to $\ll_{\mathbf{P}^a}$ are called *perfect* models of $\mathbf{P}$. □

Returning to programs $\mathbf{P}_1$ and $\mathbf{P}_2$ in (39), we can now see that they are not locally stratified with respect to Definition A.4 (while they are locally stratified with respect to the preliminary Definition A.1). Indeed, consider the following instance of the paramodulation axiom:

$$p(b) \longleftarrow p(a) \;\wedge\; (a \doteq b)$$

Here, both atoms in the premise are relevant. Therefore, we have a negative cycle in $\mathcal{D}(\mathcal{L}, \mathbf{P}_1^a)$, which goes from $\neg p(b)$ to $p(a)$, and back to $p(b)$; in $\mathcal{D}(\mathcal{L}, \mathbf{P}_2^a)$, the cycle goes from $\neg p(b)$ to $a \doteq b$, and back to $p(b)$. Similar cycles exist in the rule-dependency graphs of these programs.

Similarly, one can show that (41) has a negative cycle. Consider the following instance of the type inheritance axiom:

$$b\,[attr \Rightarrow c] \longleftarrow a\,[attr \Rightarrow c] \wedge b :: a$$

This rule belongs to the augmentation of $\mathbf{P}$ because both of its premises are relevant. Thus, $\mathbf{D}_{\mathcal{L}}(\mathbf{P}_3^a)$ has the following negative cycle: $\neg b\,[attr \Rightarrow c]$, to $p(a)$, to $a\,[attr \Rightarrow c]$, and back to $b\,[attr \Rightarrow c]$.

Proposition A.6 *Every locally stratified F-program has a unique perfect H-model.*

Proof: (Sketch) The proof is similar to that in [89]. Another, albeit indirect, proof can be obtained using Theorem 18.1. Let $\Gamma(\mathbf{P}^a)$ be a translation of $\mathbf{P}^a$ into classical logic. To prove the result, we can show that if $\prec_{\mathbf{P}^a}$ is well-founded then so is $\prec_{\Gamma(\mathbf{P}^a)}$, where the latter is the order on ground atoms that is used in the usual definition of local stratification in [89]. Then we can show that (in the notation of Theorem 18.1) Φ maps $\ll_{\mathbf{P}^a}$-minimal models of $\mathbf{P}$ into $\ll_{\Gamma(\mathbf{P}^a)}$-minimal models of $\Gamma(\mathbf{P}^a)$. □

B Appendix: Perfect Models in the Presence of Inheritance

In this appendix, we extend the results of Section 15.2.2 and Appendix A by defining perfect model semantics when negation and inheritance are working together.

The Principle of Minimally Necessary Inheritance of Section 15.2.2 will continue to guide us here. However, it needs to be extended because semantics of general logic programs relies on making negative assumptions in essential ways. Negative data are derived by "jumping to conclusions", which is a non-classical inference rule. To avoid contradiction, these negative conclusions must override inheritance just like positive facts do. Practically, this means that negative conclusions reached earlier are not to be thrown out if (under the definitions of Section 15.2.2) it becomes possible to derive positive conclusions by inheritance. Instead, inheritance is blocked if such positive conclusions are in conflict with the aforesaid negative conclusions.

This principle is realized by modifying the iterative procedure for computing the canonic model, so that it could take into account the subtleties in deriving inherited and negative facts. At present, we do not have an alternative, more declarative, characterization of the canonic model in terms of a priority relation, such as $\ll_{\mathbf{P}}$, used in Appendix A.

Stratification for Inheritance

First, we extend the rule-dependency graph of Appendix A to include a new type of arcs. The resulting *extended rule-dependency graph*, $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$ (ι symbolizes inheritance), has the same nodes as $\mathcal{D}_r(\mathcal{L}, \mathbf{P})$ and it includes all the positive and negative arcs from $\mathcal{D}_r(\mathcal{L}, \mathbf{P})$. In addition, it has *inheritance arcs*, which exist in the following cases. Let $r, r', r'' \in \mathbf{P}^*$ be such that r is of the form $\phi \longleftarrow ... \wedge \neg o[...; m@a_1, \ldots, a_n \leadsto v; ...] \wedge ...$, the rule r' is of the form $o\sharp c \longleftarrow ... \wedge \psi \wedge ...$, where $\sharp$ stands for ":" or "::", and r'' has the form $c[...; m@a_1, \ldots, a_n \leadsto v; ...] \longleftarrow \eta$. Then $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$ has two inheritance arcs, $r \overset{\iota}{\longleftarrow} r'$ and $r \overset{\iota}{\longleftarrow} r''$.

An *inheritance-negative path* (or just *inheritance path*, for brevity) in $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$ is a path that contains at least one negative arc or an inheritance arc. A *negative path* is a path that contains at least one negative arc and no inheritance arcs (*i.e.*, only negative and positive arcs). Thus, a negative path is also an inheritance path, but the opposite is not always true.

The new rule-dependency graph, $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$, is used to extend the relations $\prec_{\mathbf{P}}$ and $\preceq_{\mathbf{P}}$ that exist over the clauses in $\mathbf{P}^*$ (defined in Appendix A). These new orderings lead to a modification of the notion of stratification that directly affects program stratas, which are defined in (37).

For any pair of rules, $r, r' \in \mathbf{P}^*$, we shall write:

$r \prec_{\mathbf{P}}^\iota r'$ if $r \prec_{\mathbf{P}} r'$, *i.e.*, a negative path in $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$ goes from r to r',
or if an inheritance path that is not part of a cycle in $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$ goes from r to r';
$r \preceq_{\mathbf{P}}^\iota r'$ if a path in $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P})$ goes from r to r' and no *negative* path goes from r' to r.

Since $\prec_{\mathbf{P}}^\iota$ and $\preceq_{\mathbf{P}}^\iota$ take inheritance paths into consideration, they make more rules comparable than do $\prec_{\mathbf{P}}$ and $\preceq_{\mathbf{P}}$. Also note that the new preorders extend the old ones:

$$r \prec_{\mathbf{P}} r' \text{ implies } r \prec_{\mathbf{P}}^\iota r' \quad \text{and} \quad r \preceq_{\mathbf{P}} r' \text{ implies } r \preceq_{\mathbf{P}}^\iota r' \tag{42}$$

Stratified programs are defined similarly to Definition A.1, except that we now use the extended ordering, $\prec_{\mathbf{P}^a}^\iota$, instead of $\prec_{\mathbf{P}}$. Also, as explained in Appendix A, one should really work with augmented programs, $\mathbf{P}^a$, so that the equality predicate will be accounted for.

Definition B.1 (Local Stratification for Inheritance) A program, $\mathbf{P}$, is *locally stratified for inheritance* (or *locally ι-stratified*) if $\prec_{\mathbf{P}^a}^\iota$ is a well-founded relation on $(\mathbf{P}^a)^*$. □

Note that, due to (42), ι-stratified programs are also stratified in the earlier sense. However, stratified programs may not be ι-stratified. To see this, consider the following program, $\mathbf{P}$:

$$\begin{array}{ll} r: & cl[attr \bullet\!\!\to X] \longleftarrow \neg o[attr \to f(X)] \\ r': & o : cl \end{array}$$

Clearly, this program is locally stratified according to Definition A.4. To show that $\mathbf{P}$ is not ι-stratified, let r_t denote the instance of the above rule r in which X is replaced by a term, t. For example, $r_{f(a)}$ is as follows:

$$cl[attr \bullet\!\!\to f(a)] \longleftarrow \neg o[attr \to f^2(a)]$$

It is easy to see that $\mathcal{D}_r^\iota(\mathcal{L}, \mathbf{P}^a)$ has an inheritance arc $r_{f^{n-1}(a)} \xleftarrow{\iota} r_{f^n(a)}$, for every $n > 0$, and no path goes the other way. Therefore, we have an infinite chain $\cdots \prec_{\mathbf{P}^a}^\iota r_{f^2(a)} \prec_{\mathbf{P}^a}^\iota r_{f(a)} \prec_{\mathbf{P}^a}^\iota r_a$, which shows that $\prec_{\mathbf{P}^a}^\iota$ is not well-founded and, thus, $\mathbf{P}$ is not locally stratified for inheritance.

Note that inheritance cycles (even the ones that go through negative arcs) do not necessarily turn programs into unstratified ones. This is seen from the examples at the end of this appendix.

Perfect Models Defined

As mentioned earlier, we need to modify the definition of triggers to protect negative assumptions made during the computation of perfect models in (38). For this, we need to modify our definitions in order to make these assumptions explicit.

Let $\mathbf{M}$ be an H-structure and Δ be a set of ground F-atoms. A *new-style trigger* in $\mathbf{M}$ under the set of negative assumptions Δ is a trigger, τ (in the sense of Definition 15.1), such that $\tau(\mathbf{M}) \cap \Delta$ is empty, *i.e.*, firing τ does not contradict the negative assumptions in Δ.

In this appendix, we shall deal only with new-style triggers, and only this kind of triggers will be used by the operators $\Im_{\mathbf{P}}^{\mathcal{T}}$ and $\Im_{\mathbf{P}}^{*}$ (defined in Section 15.2.2).

We are now ready to define perfect models in the presence of inheritance, the ι-perfect models. The definition combines the $\Im_{\mathbf{P}}^{*}$ operator (modified with new-style triggers) with the iterative procedure (38) of Appendix A.

Let $\mathbf{P}$ be a locally ι-stratified program and let $\mathbf{P}_1^*, \mathbf{P}_2^*, \ldots$ be a stratification of $\mathbf{P}^*$ (defined as in (37) except that $\prec_{\mathbf{P}^a}^{\iota}$ and $\preceq_{\mathbf{P}^a}^{\iota}$ are used instead of $\prec_{\mathbf{P}}$ and $\preceq_{\mathbf{P}}$). Then *ι-perfect models* of $\mathbf{P}$ are defined as the end-product of the following iteration:

$$\begin{array}{lcll}
\mathbf{M}_1 & = & T_{\mathbf{P}_1^*} \uparrow \omega(\emptyset) & \\
\mathbf{M}_1^{\iota} & = & \Im_{\mathbf{P}_1^*}^{*}(\mathbf{M}_1) & \text{where triggers are subject to negative assumptions in } \Delta_1 \\
\mathbf{M}_2 & = & T_{\mathbf{P}_2^*} \uparrow \omega(\mathbf{M}_1^{\iota}) \bigcup \mathbf{M}_1^{\iota} & \\
\mathbf{M}_2^{\iota} & = & \Im_{\mathbf{P}_2^*}^{*}(\mathbf{M}_2) & \text{where triggers are subject to negative assumptions in } \Delta_2 \\
\mathbf{M}_3 & = & T_{\mathbf{P}_3^*} \uparrow \omega(\mathbf{M}_2^{\iota}) \bigcup \mathbf{M}_2^{\iota} & \\
\mathbf{M}_3^{\iota} & = & \Im_{\mathbf{P}_3^*}^{*}(\mathbf{M}_3) & \text{where triggers are subject to negative assumptions in } \Delta_3 \\
\ddots & & & \\
\mathbf{M} & = & \bigcup_{i=1}^{\infty} \mathbf{M}_i^{\iota} &
\end{array} \tag{43}$$

Since the operator $\Im_{\mathbf{P}}^{*}$ is non-deterministic, there may be more than one ι-perfect model. The negative assumptions, Δ_i, are constructed as follows. The first set, Δ_1, is empty. For $i > 1$, we define:

$$\Delta_i \;=\; \Delta_{i-1} \cup \{ l \mid \text{ there is } h \longleftarrow \mathit{body} \in \mathbf{P}_i^* \text{ such that } \mathbf{M}_i \models \mathit{body} \text{ and } \neg l \text{ occurs in } \mathit{body} \}$$

In other words, the negative assumptions at stage i consist of the assumptions made previously plus the atoms that occur negatively in the bodies of active rules in $\mathbf{P}_i^*$.

Examples

We illustrate the above concepts using a number of examples. The first program espouses a somewhat outdated ideology, but it is good for illustrating the technical idea. It says, if managers are paid well then employees are paid poorly:

$$\begin{array}{ll}
r_1 & \mathit{manager} :: \mathit{empl} \\
r_2 & \mathit{empl}[\mathit{salary} \bullet\!\!\to \mathit{low}] \longleftarrow \neg \mathit{manager}[\mathit{salary} \bullet\!\!\to \mathit{low}]
\end{array} \tag{44}$$

Here we have the arc $r_2 \overset{\iota}{\longleftarrow} r_1$ and so $r_1 \prec_{\mathbf{P}}^{\iota} r_2$. Thus, only *manager* :: *employee* is computed in the first stratum. In the second stratum, we make a negative assumption, $\Delta_2 = \{\mathit{manager}[\mathit{salary} \bullet\!\!\to \mathit{low}]\}$, which leads to a derivation of $\mathit{employee}[\mathit{salary} \bullet\!\!\to \mathit{low}]$ in that stratum. At this point, according to Definition 15.1, *manager* could inherit $\mathit{salary} \bullet\!\!\to \mathit{low}$. However, $\langle \mathit{manager} :: \mathit{employee}, \mathit{salary} \bullet\!\!\to \rangle$ is not a new-style trigger (due to the above negative assumption), so inheritance does not take place.

The next program is a bit more involved (its political message was not yet finalized at the time of this writing):

$$\begin{array}{ll}
r_1 & p : q \\
r_2 & q[\mathit{attr} \bullet\!\!\to a] \\
r_3 & r[\mathit{attr} \to b] \longleftarrow \neg p[\mathit{attr} \to a] \\
r_4 & t[\mathit{attr} \to c] \longleftarrow \neg r[\mathit{attr} \to b]
\end{array}$$

Here, $r_1, r_2 \prec^{\iota}_{\mathbf{P}} r_3 \prec^{\iota}_{\mathbf{P}} r_4$. Therefore, first, p inherits $attr \rightarrow a$, which blocks r_3 from firing. As a result, $r[attr \rightarrow b]$ is not derived and so $t[attr \rightarrow c]$ is derived in the second stratum.

For a more involved example, consider this:

$$\begin{array}{ll} r_1 & a : b \\ r_2 & p : d \\ r_3 & b[attr' \bullet\!\!\twoheadrightarrow c] \longleftarrow \neg p[attr \twoheadrightarrow e] \\ r_4 & d[attr \bullet\!\!\twoheadrightarrow e] \longleftarrow \neg a[attr' \twoheadrightarrow c] \end{array} \tag{45}$$

Here $r_1, r_2 \prec^{\iota}_{\mathbf{P}} r_3, r_4$ and $r_3 \preceq^{\iota}_{\mathbf{P}} r_4 \preceq^{\iota}_{\mathbf{P}} r_3$. Note that r_3 and r_4 are involved in an inheritance cycle in the extended rule-ordering graph. However, this does not make the program unstratified. Indeed, r_3 and r_4 are incomparable with respect to $\prec^{\iota}_{\mathbf{P}}$, so $\prec^{\iota}_{\mathbf{P}}$ is well-founded.

In this example, $a : b$ and $p : d$ are computed first. Then we apply the generalized one-step inference operator to compute the second stratum. To this end, we make two negative assumptions, $\Delta_2 = \{p[attr \twoheadrightarrow e],\ a[attr \twoheadrightarrow c]\}$, which enables the derivation of $b[attr \bullet\!\!\twoheadrightarrow c]$ and $d[attr \bullet\!\!\twoheadrightarrow e]$. Note that Definition 15.1 sanctions two inheritance triggers, $\langle a : b,\ attr \bullet\!\!\twoheadrightarrow \rangle$ and $\langle p : d,\ attr \bullet\!\!\twoheadrightarrow \rangle$. However, these are not new-style triggers, because they are in conflict with the negative assumptions in Δ_2.

The next example is a feeble attempt to capture a phenomenon from the academic world:

$$\begin{array}{ll} r_1 & theoretician :: professor \\ r_2 & joe : professor \\ r_3 & theoretician[funding \bullet\!\!\rightarrow low] \\ r_4 & X : theoretician \longleftarrow \neg X[programmer \rightarrow good] \\ r_5 & X[travels \rightarrow much] \longleftarrow \neg X[funding \rightarrow low] \end{array} \tag{46}$$

The thing to note here is that $r_4 \prec^{\iota}_{\mathbf{P}} r_5$ (when X is instantiated to joe) and, thus, r_4 should be applied before r_5. After computing the two facts, r_1 and r_2, we jump to a negative conclusion, $\Delta_2 = \{joe[programmer \rightarrow good]\}$, and derive $joe : theoretician$ via rule r_4. This activates the trigger $\langle joe : theoretician,\ funding \bullet\!\!\rightarrow \rangle$, which sanctions the inheritance of $joe[funding \rightarrow low]$. This newly inferred atom prevents rule r_5 from firing, and the computation terminates.

C Appendix: A Unification Algorithm for F-molecules

This appendix presents an algorithm for finding a complete set of mgu's for a pair of molecules. We remind (from Section 11.1) that all complete sets of mgu's are equivalent to each other and, therefore, it suffices to find just one such set. Recall that a molecule can be represented as a conjunction of its constituent atoms. For a pair of molecules, T_1 and T_2, an mgu of T_1 *into* T_2 is a *most general* substitution that turns every constituent atom of T_1 into a constituent atom of T_2. Different correspondences between the constituent atoms of T_1 and T_2 may lead to different mgu's. To find a *complete* set of mgu's, we must take all such correspondences into account. Let φ be an atom of either of the following forms:

(i) $P\,[Method \,@\, Q_1, \ldots, Q_k \leadsto R]$, where $\leadsto$ denotes any one of the six types of arrows allowed in method expressions: $\rightarrow$, $\bullet\!\!\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!\!\!\Rightarrow$;

(ii) $P\,[Method \,@\, Q_1, \ldots, Q_k \leadsto \{\,\}]$, where $\leadsto$ denotes $\twoheadrightarrow$ or $\bullet\!\!\twoheadrightarrow$; or

(iii) $P\,[Method \,@\, Q_1, \ldots, Q_k \leadsto (\,)]$, where $\leadsto$ denotes $\Rightarrow$ or $\Rightarrow\!\!\!\!\Rightarrow$.

```
Input :   A Pair of molecules, $T_1$ and $T_2$.
Output :  $\Omega$ — a complete set of mgu's of $T_1$ into $T_2$.

  1. If $id(T_1)$ and $id(T_2)$ are unifiable
     then $\theta := UNIFY(< id(T_1) >, < id(T_2) >)$.
     else Stop: $T_1$ and $T_2$ are not unifiable.

  2. If $T_1$ is of the form $S\,[\,]$ (a degenerated molecule) then Stop: $\theta$ is the only mgu.

  3. $\Omega := \{\}$.

  4. for each mapping $\lambda \in Map(T_1, T_2)$ do:
        $\sigma_\lambda := \theta$.       /* $\sigma_\lambda$ is a variable used here to hold a potential mgu */
        for each atom $\varphi$ in $atoms(T_1)$ do:
           Let $\psi$ be $\lambda(\varphi)$.
           Unify tuples $\sigma_\lambda(\vec{S_1})$ and $\sigma_\lambda(\vec{S_2})$, where
              $\vec{S_1} = < method(\varphi), arg_1(\varphi), \ldots, arg_n(\varphi), val(\varphi) >$ and
              $\vec{S_2} = < method(\psi), arg_1(\psi), \ldots, arg_n(\psi), val(\psi) >$.
           /* if $val(\varphi)$ or $val(\psi)$ is $\emptyset$, they are treated as constant $\emptyset$ for the unification purposes */
           If $\sigma_\lambda(\vec{S_1})$ and $\sigma_\lambda(\vec{S_2})$ unify then $\sigma_\lambda := UNIFY(\sigma_\lambda(\vec{S_1}), \sigma_\lambda(\vec{S_2})) \circ \sigma_\lambda$.
           else Discard this $\sigma_\lambda$ and jump out of the inner for each to select another $\lambda$.
           $\Omega := \Omega \cup \{\sigma_\lambda\}$.
        endfor
     endfor

  5. Return $\Omega$, a complete set of mgu's of $T_1$ into $T_2$.
```

Figure 10: Computing a Complete Set of MGU's

The following notation is used in the unification algorithm in Figure 10, where ϕ is as above:

$$\begin{aligned} id(\varphi) &= P \\ method(\varphi) &= Method \\ arg_i(\varphi) &= Q_i, \text{ for } i = 1, \ldots, k \end{aligned}$$

$$val(\varphi) = \begin{cases} R & \text{if } \varphi \text{ is of the form (i) above} \\ \emptyset & \text{if } \varphi \text{ has the form (ii) or (iii) above} \end{cases}$$

In addition, if T is a molecule then $atoms(T)$ will denote the set of atoms of T. If T_1 and T_2 are molecules then $Map(T_1, T_2)$ denotes the collection of mappings $\{\lambda : atoms(T_1) \longrightarrow atoms(T_2)\}$ that preserve method arities and the type of the method expression (*i.e.*, the type of the arrow used in those expressions).

As noted in Section 11.1, the mgu of a pair of tuples of id-terms, $\langle P_1, \ldots, P_n \rangle$ and $\langle Q_1, \ldots, Q_n \rangle$, coincides with the mgu of the terms $f(P_1, \ldots, P_n)$ and $f(Q_1, \ldots, Q_n)$, where f is an arbitrary n-ary function symbol. So, we can use any standard unification procedure to unify tuples of such id-terms. Given a pair of tuples of id-terms, $\vec{S_1}$ and $\vec{S_2}$, we use $UNIFY(\vec{S_1}, \vec{S_2})$ to denote a procedure that returns the mgu of $\vec{S_1}$ and $\vec{S_2}$, if one exists.

Lemma C.1 *The algorithm in Figure 10 correctly finds a complete set of mgu's of T_1 into T_2.*

Proof: Clearly, all elements of Ω are mgu's of T_1 into T_2, so we will only show that Ω is complete. Consider a unifier θ of T_1 into T_2. By definition, there is a mapping $\lambda \in Map(T_1, T_2)$ from $atoms(T_1)$ to $atoms(T_2)$, such that θ maps every φ in $atoms(T_1)$ into an appropriate atom $\lambda(\varphi)$ in $atoms(T_2)$. Clearly, substitution σ_λ constructed in the inner **for**-loop in Step 4 of the above algorithm is a most general unifier that maps every $\varphi \in atoms(T_1)$ to $\lambda(\varphi) \in atoms(T_2)$. So, θ is an instance of σ_λ.

Summarizing, we have shown that 1) each element of Ω is an mgu; and 2) if θ is a unifier of T_1 into T_2, then θ is an instance of an element $\sigma_\lambda \in \Omega$. Hence, Ω is a complete set of mgu's of T_1 into T_2. □

It is not hard to see that the unification problem for F-molecules is polynomial time isomorphic to the so-called *first-order subsumption problem*, which is known to be NP-complete [44].[34] It is therefore not surprising that the worst-case complexity of the algorithm in Figure 10 is exponential, and so is the worst-case size of the set of unifiers. This complexity is due to the following two factors:

1. *Unification of sets:*
   This problem is not specific to F-logic. Every language that allows the use of sets in any essential way has to put up with the exponential worst-case complexity of set-unification.

2. *Permutation of methods inside molecules:*
   Since methods may be denoted by *nonground* id-terms, they can match each other in many different ways. For instance, in unifying $P[X \to V;\ Y \to W]$ and $P[X' \to V';\ Y' \to W']$, the method denoted by X can match either X' or Y'; similarly for Y. Thus, added complexity should be expected due to the higher-order syntax of F-logic.

The key factor in estimating the complexity of unification in "practical" cases is the number of atoms comprising each molecule in the bodies of the rules. For, if n_1 is the number of atoms in T_1 and n_2 is the same for T_2, then the number of unifiers of T_1 into T_2 is bounded by $n_2^{n_1}$. Now, to resolve a pair of rules $\ldots \longleftarrow \ldots, T_1, \ldots$ and $T_2 \longleftarrow \ldots,$ where T_1 and T_2 are data or signature molecules, we need to unify T_1 into T_2 and, therefore, the above parameter, n_1, which denotes the number of atoms in T_1, is most crucial. However, our experience with F-programs and the examples in this paper show that this number is usually very small (≤ 2). So, we believe, combinatorial explosion is unlikely to happen in practice.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for manipulation of complex objects. Technical report, Rapport de Recherche INRIA, 1988. To appear in TODS.

[2] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Workshop on Database Programming Languages*, pages 253–276, Roscoff, France, September 1987.

[3] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD Conference on Management of Data*, pages 159–173, 1989.

[4] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *ACM SIGMOD Conference on Management of Data*, pages 32–41, May 1993.

[34]First-order subsumption is a problem of finding a most general substitution that turns one set of first-order terms into a subset of another such set.

[5] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 240–250, 1988.

[6] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[7] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[8] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, August 1993.

[9] R. Anderson and W.W. Bledsoe. A linear format resolution with merging and a new technique for establishing completeness. *Journal of ACM*, 17(3):525–534, 1970.

[10] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 40–57, 1989.

[11] M. Balaban. The generalized concept formalism – A frames and logic based representation model. In *Proceedings of the Canadian Artificial Intelligence Conference*, pages 215–219, 1986.

[12] M. Balaban and S. Strack. LOGSTER — A relational, object-oriented system for knowledge representation. In *Intl. Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 210–219, 1988.

[13] F. Bancilhon. Object-oriented database systems. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 152–162, 1988.

[14] F. Bancilhon and S.N. Khoshafian. A calculus of complex objects. *Journal of Computer and System Sciences*, 38(2):326–340, April 1989.

[15] J. Banerjee, W. Kim, and K.C. Kim. Queries in object-oriented databases. In *Proc. of the 4-th Intl. Conf. on Data Engineering*, Los Angeles, CA, February 1988.

[16] C. Beeri. Formal models for object-oriented databases. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 370–395. Elsevier Science Publ., 1989.

[17] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL). Technical report, MCC, 1987.

[18] C. Beeri, R. Nasr, and S. Tsur. Embedding ψ-terms in a horn-clause logic language. In *Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 347–359. Morgan Kaufmann, 1988.

[19] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, April 1991.

[20] H.A. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[21] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-270, University of Toronto, April 1992. Revised: February 1994. Available in *csri-technical-reports/270/report.ps* by anonymous ftp to *csri.toronto.edu.*

[22] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming (ICLP)*, Budapest, Hungary, June 1993.

[23] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[24] S. Brass and U.W. Lipeck. Semantics of inheritance in logical object specifications. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 411–430, December 1991.

[25] G. Brewka. The logic of inheritance in frame systems. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 483–488, 1987.

[26] M. Bugliesi and H.M. Jamil. A logic for encapsulation in object-oriented languages. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, pages 215–229, Madrid, Spain, September 1994. Springer-Verlag.

[27] P. Buneman and R.E. Frankel. FQL — A functional query language. In *ACM SIGMOD Conference on Management of Data*, pages 52–58, 1979.

[28] P. Buneman and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 1989.

[29] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2):138–164, February 1988.

[30] M. Carey, D. DeWitt, and S. Vanderberg. A data model and query language for EXODUS. In *ACM SIGMOD Conference on Management of Data*, 1988.

[31] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[32] W. Chen. A theory of modules based on second-order logic. In *IEEE Symposium on Logic Programming (SLP)*, pages 24–33, September 1987.

[33] W. Chen and M. Kifer. Polymorphic types in higher-order logic programming. Technical Report 93/20, Department of Computer Science, SUNY at Stony Brook, December 1993.

[34] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[35] W. Chen and D.S. Warren. C-logic for complex objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 369–378, March 1989.

[36] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[37] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 125–136, 1990.

[38] S.W. Dietrich. Extension tables: Memo relations in logic programming. In *IEEE Symposium on Logic Programming (SLP)*, pages 264–273. IEEE, September 1987.

[39] G. Dobbie and R. Topor. On the declarative and procedural semantics of deductive object-oriented systems. *Journal of Intelligent Information Systems*, 4, February 1995.

[40] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[41] D.W. Etherington and R. Reiter. On inheritance hierarchies with exceptions. In *National Conference on Artificial Intelligence (AAAI)*, pages 104–108, Washington, D.C., 1983.

[42] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of ACM*, 28(9):904–920, 1985.

[43] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 273–284, Santiago, Chile, 1994.

[44] M.R. Garey and Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company, San Francisco, CA, 1978.

[45] H. Geffner and T. Verma. Inheritance = chaining + defeat. In *Intl. Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 411–418, 1989.

[46] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[47] R.P. Hall. Computational approaches to analogical reasoning: A comparative study. *Artificial Intelligence*, 30:39–120, 1989.

[48] P.J. Hayes. The logic for frames. In D. Metzing, editor, *Frame Conception and Text Understanding*, pages 46–61. Walter de Gruyter and Co., 1979.

[49] P. Hill and R. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. The MIT Press, 1992.

[50] J.F. Horty, R.H. Thomason, and D.S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic nets. In *National Conference on Artificial Intelligence (AAAI)*, pages 358–363, 1987.

[51] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 455–468, Brisbane, Australia, 1990.

[52] F.N. Kesim and M. Sergot. On the evolution of objects in a logic programming framework. In *Proceedings of the Intl. Conference on Fifth Generation Computer Systems*, pages 1052–1060, Tokyo, Japan, June 1992.

[53] S.N. Khoshafian and G.P. Copeland. Object identity. In *Proceedings of the International Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 406–416, 1986.

[54] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *ACM SIGMOD Conference on Management of Data*, pages 393–402, June 1992.

[55] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and schema. In *ACM SIGMOD Conference on Management of Data*, pages 134–146, 1989.

[56] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, November 1992.

[57] M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–368, April 1992.

[58] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 379–393, March 1989.

[59] M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *Intl. Symposium on Logic in Computer Science (LICS)*, pages 310–321, Amsterdam, The Netherlands, July 1991. Expanded version: TR 90/23 under the same title, Department of Computer Science, University at Stony Brook, July 1990.

[60] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77–120, August 1993.

[61] W. Kim, J. Banerjee, H-T. Chou, J.F. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *Proceedings of the International Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, 1987.

[62] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 231–239, 1988.

[63] R. Krishnamurthy and S. Naqvi. Towards a real horn clause language. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1988.

[64] T. Krishnaprasad, M. Kifer, and D.S. Warren. On the circumscriptive semantics of inheritance networks. In *Intl. Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 448–457, 1989.

[65] T. Krishnaprasad, M. Kifer, and D.S. Warren. On the declarative semantics of inheritance networks. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 1099–1103, 1989.

[66] G. Kuper and M.Y. Vardi. A new approach to database logic. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1984.

[67] G.M. Kuper. An extension of LPS to arbitrary sets. Technical report, IBM, Yorktown Heights, 1987.

[68] G.M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, August 1990.

[69] E. Laenens, D. Sacca, and D. Vermeir. Extending logic programming. In *ACM SIGMOD Conference on Management of Data*, pages 184–193, June 1990.

[70] E. Laenens and D. Vermeir. A fixpoint semantics for ordered logic. *Journal Logic and Computation*, 1(2):159–185, 1990.

[71] C. Lecluse and P. Richard. The O_2 database programming language. In *Intl. Conference on Very Large Data Bases (VLDB)*, August 1989.

[72] J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, 1987.

[73] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.

[74] D. Maier. Why database languages are a bad idea (position paper). In *Proc. of the Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[75] D. Maier. Why isn't there an object-oriented data model. Technical report, Oregon Graduate Center, May 1989.

[76] F.G. McCabe. *Logic and Objects*. Prentice Hall International, London, England, 1992.

[77] J. McCarthy. First order theories of individual concepts and propositions. In J.E. Hayes and D. Michie, editors, *Machine Inteligence*, volume 9, pages 129–147. Edinburgh University Press, 1979.

[78] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[79] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[80] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind design*, pages 95–128. MIT Press, Cambridge, MA, 1981.

[81] P. Mishra. Towards a theory of types in Prolog. In *IEEE Symposium on Logic Programming (SLP)*, pages 289–298, 1984.

[82] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 109–124, 1990.

[83] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Van Gelder. YAWN! (Yet another window on NAIL!). *IEEE Database Engineering*, 6:211–226, 1987.

[84] A. Motro. BAROQUE: A browser for relational databases. *ACM Transactions on Office Information Systems*, 4(2):164–181, 1986.

[85] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[86] E. Neuhold and M. Stonebraker. Future directions in DBMS research (The Laguna Beech report). *SIGMOD Record*, 18(1), March 1989.

[87] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[88] H. Przymusinska and M. Gelfond. Inheritance hierarchies and autoepistemic logic. In *Intl. Symposium on Methodologies for Intelligent Systems (ISMIS)*, 1989.

[89] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[90] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 11–21, 1989.

[91] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *IEEE Symposium on Logic Programming (SLP)*, pages 140–159, 1988.

[92] K.A. Ross. Relations with relation names as arguments: Algebra and calculus. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 346–353, May 1992.

[93] M.A. Roth, H.F. Korth, and D.S. Batory. SQL/NF: A query language for $\neg$1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[94] J.W. Schmidt. Some high-level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[95] D.W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, pages 140–173, 1981.

[96] M. Stefik and D.G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, pages 40–62, January 1986.

[97] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.

[98] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Intl. Conference on Logic Programming (ICLP)*, pages 84–98, 1986.

[99] K. Thirunarayan and M. Kifer. A theory of nonmonotonic inheritance based on annotated logic. *Artificial Intelligence*, 60(1):23–50, March 1993.

[100] D.S. Touretzky. *The Mathematics of Inheritance.* Morgan-Kaufmann, Los Altos, CA, 1986.

[101] D.S. Touretzky, J.F. Horty, and R.H. Thomason. A clash of intuitions: The current state of nonmonotonic multiple inheritance systems. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 476–482, 1987.

[102] J.D. Ullman. Database theory: Past and future. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–10, 1987.

[103] J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, 1988.

[104] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–10, 1989.

[105] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[106] D.S. Warren. Memoing for logic programming. *Communications of ACM*, 35(3):93–111, March 1992.

[107] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. MIT Press, 1987.

[108] J. Wu. *A Theory of Types and Polymorphism in Logic Programming.* PhD thesis, SUNY at Stony Brook, 1992.

[109] J. Xu. *A Theory of Types and Type Inference in Logic Programming Languages.* PhD thesis, SUNY at Stony Brook, 1989.

[110] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphically typed logic programs. In *Intl. Conference on Logic Programming (ICLP)*, Paris, France, June 1991.

[111] E. Yardeni and E. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog*, volume 2. MIT Press, 1987.

[112] C. Zaniolo. The database language GEM. In *ACM SIGMOD Conference on Management of Data*, pages 423–434, 1983.

[113] C. Zaniolo, H. Aït-Kaci, D. Beech, S. Cammarata, L. Kerschberg, and D. Maier. Object-oriented database and knowledge systems. Technical Report DB-038-85, MCC, 1985.